Teoría sintáctico gramatical de objetos

diseño de sistemas informáticos orientados a objetos desde el lenguaje natural

con ejemplos en
PYTHON & PHP

Eugenia Bahit

Teoría sintáctico-gramatical de Objetos

Diseño evolucionado de sistemas informáticos orientados a objetos desde el lenguaje natural, con ejemplos en Python y PHP

Eugenia Bahit

Impreso en España por Bubok Publishing S.L.

A mis alumnos, quienes día a día llenan mis tardes de esperanzas; a Richard Stallman, por haberme enseñado el verdadero valor del conocimiento y a todas aquellas personas que disfrutan jugando con el ingenio y compartiendo su sabiduría...

Happy Hacking!

Contenidos

Capítulo I: Introducción informal para perderle el miedo a la orientación a objetos....9
¿ Cómo pensar en objetos?....9
Entendiendo el paradigma de la programación orientada a objetos....20
Elementos y Características de la POO....21
Capítulo II: Teoría sintáctico-gramatical de objetos y el diseño de objetos en lenguaje natural....25
Del lenguaje natural al lenguaje de programación....35
Capítulo III: Creación de objetos y acceso a propiedades....39
Capítulo IV: Patrón de diseño compuesto (composite pattern)....45
Introducción a la Arquitectura de Software....45
Atributos de calidad....46
Niveles de abstracción....48
Estilo Arquitectónico....49
Patrón Arquitectónico....50
Patrón de Diseño....51
Composite design pattern (patrón de diseño compuesto)....52
Capítulo V: Objetos compositores exclusivos, identidad, pertenencia y agregación....57
Capítulo VI: Objetos relacionales simples -o multiplicadores-....67
Capítulo VII: Mapeo relacional de objetos y estructuras de almacenamiento de datos sustentables....73
Capítulo VIII: Objetos, subtipos y herencia real....83
Capítulo IX: Modelado de objetos y agrupación....89
Capítulo X: Bases de datos, consultas y abstracción....91
Creando una capa de abstracción en Python....92
Creando una capa de abstracción en PHP con mysqli....94
Recetario....95
Capítulo XI: El método save() en objetos simples, compuestos y relacionales....107
Capítulo XII: El método destroy()....119
Capítulo XIII: Métodos get() estándar, para objetos simples y objetos compuestos....121
Capítulo XIV: Los métodos get() en objetos compositores de

pertenencia..127
Capítulo XV: El método get() de los objetos relacionales multiplicadores..133
Capítulo XVI: Factoría de objetos con Factory Pattern. Objetos compuestos con métodos get() mucho más simples..............137
Capítulo XVII: Objetos colectores de instancia única y Singleton Pattern..143
Características de un Singleton colector en PHP..............145
Características de un Singleton colector en Python..........147
El método get() del singleton colector............................151
Capítulo XVIII: Objetos relacionales complejos (conectores lógicos relacionales)..153
Los métodos save(), get() y destroy() del conector lógico157

Capítulo I: Introducción informal para perderle el miedo a la orientación a objetos

La orientación a objetos es un paradigma de programación que puede resultar complejo, si no se lo interpreta de forma correcta desde el inicio. Por eso, en esta primera parte, nos enfocaremos primero, en cuestiones de conceptos básicos, para luego, ir introduciéndonos de a poco, en principios teóricos elementalmente necesarios para implementar la orientación a objetos en la práctica.

¿ Cómo pensar en objetos?

Pensar en objetos, puede resultar -al inicio- una tarea difícil. Sin embargo, difícil no significa complejo. Por el contrario, pensar en objetos representa la mayor simplicidad que uno podría esperar del mundo de la programación. Pensar en objetos, es simple... aunque lo simple, no

necesariamente signifique sencillo.

Y ¿qué es un objeto?

Pues, como dije antes, es “simple”. Olvidemos los formalismos, la informática y todo lo que nos rodea. Simplemente, olvida todo y concéntrate en lo que sigue. Lo explicaré de manera “simple”:

Un objeto es “una cosa”. Y, si una cosa es un sustantivo, entonces un objeto es un sustantivo.

Mira a tu alrededor y encontrarás decenas, cientos de objetos. Tu ordenador, es un objeto. Tú, eres un objeto. Tu llave es un objeto. El cenicero (ese que tienes frente a ti cargado de colillas de cigarrillo), es otro objeto. Tu mascota también es un objeto.

“Cuando pensamos en “objetos”, todos los sustantivos son objetos.”

Sencillo ¿cierto? Entonces, de ahora en más, solo concéntrate en pensar la vida en objetos (al menos, hasta terminar de leer este libro).

Ahora ¿qué me dices si describimos las cualidades de un objeto?

Describir un objeto, es simplemente mencionar sus

cualidades. Las cualidades son adjetivos.

Podemos decir que un adjetivo es una cualidad del sustantivo.

Entonces, para describir “la manera de ser” de un objeto, debemos preguntarnos ¿cómo es el objeto? Toda respuesta que comience por “el objeto es”, seguida de un adjetivo, será una cualidad del objeto.

Algunos ejemplos:

- El objeto es verde
- El objeto es grande
- El objeto es feo

Ahora, imagina que te encuentras frente a un niño de 2 años (niño: objeto que pregunta cosas que tú das por entendidas de forma implícita). Y cada vez que le dices las cualidades de un objeto al molesto niño-objeto, éste te pregunta: -”¿Qué es...?”, seguido del adjetivo con el cuál finalizaste tu frase. Entonces, tu le respondes diciendo “es un/una” seguido de un sustantivo. Te lo muestro con un ejemplo:

- El objeto es verde. ¿Qué es verde? Un color.
- El objeto es grande. ¿Qué es grande? Un tamaño.
- El objeto es feo. ¿Qué es feo? Un aspecto.

Estos sustantivos que responden a la pregunta del niño, pueden pasar a formar parte de una locución adjetiva que especifique con mayor precisión, las descripciones anteriores:

- El objeto es de color verde.
- El objeto es de tamaño grande.
- El objeto es de aspecto feo.

Podemos decir entonces -y todo esto, gracias al niño-objeto-, que una cualidad, es un atributo (derivado de "cualidad atribuible a un objeto") y que entonces, un objeto es un sustantivo que posee atributos, cuyas cualidades lo describen.

Algunos objetos, también se componen de otros objetos...

Además de cualidades (locución adjetiva seguida de un adjetivo), los objetos "tienen otras cosas".

Estas “otras cosas”, son aquellas “pseudo-cualidades” que en vez de responder a ¿cómo es el objeto? responden a “¿cómo está compuesto el objeto?” o incluso, aún más simple “¿Qué tiene el objeto?”.

La respuesta a esta pregunta, estará dada por la frase “el objeto tiene...”, seguida de un adjetivo cuantitativo o cuantificador (uno, varios, muchos, algunos, unas cuantas) más un sustantivo.

Algunos ejemplos:

- El objeto tiene algunas antenas
- El objeto tiene un ojo
- El objeto tiene unos cuantos pelos

Los componentes de un objeto, también integran los atributos de ese objeto. Solo que estos atributos, son algo particulares: son otros objetos que poseen sus propias cualidades. Es decir, que estos “atributos-objeto” también responderán a la pregunta “¿Cómo es/son ese/esos/esas?” seguido del atributo-objeto (sustantivo).

Ampliemos el ejemplo para que se entienda mejor:

- El objeto tiene algunas antenas. ¿Cómo son esas antenas?
  - Las antenas son de color violeta
  - Las antenas son de longitud extensa
- El objeto tiene un ojo. ¿Cómo es ese ojo?
  - El ojo es de forma oval
  - El ojo es de color azul
  - El ojo es de tamaño grande
- El objeto tiene unos cuantos pelos. ¿Cómo son esos pelos?
  - Los pelos son de color fucsia
  - Los pelos son de textura rugosa

Podemos decir entonces, que **un objeto puede tener dos tipos de atributos:**

1. Los que responden a la pregunta *"¿Cómo es el objeto?"* con la frase *"El objeto es..."* + locución adjetiva + adjetivo (atributos

definidos por cualidades)

2. Los que responden a la pregunta *“¿Qué tiene el objeto?”* con la frase *“El objeto tiene...”* + adjetivo cuantitativo (cantidad) + sustantivo (atributos definidos por las cualidades de otro objeto)

Hay objetos que comparten características con otros objetos

Resulta ser, que nuestro Objeto, es prácticamente igual a un nuevo objeto. Es decir, que el nuevo objeto que estamos viendo, tiene absolutamente todas las características que nuestro primer objeto. Es decir, tiene los mismos atributos. Pero también, tiene algunas más. Por ejemplo, este nuevo objeto, además de los atributos de nuestro primer objeto, tiene un pie. Es decir, que las características de nuestro nuevo objeto, serán todas las del objeto original, más una nueva: pie.

Repasemos las características de nuestro nuevo objeto:

- El nuevo objeto es de color verde.

- El nuevo objeto es de tamaño grande.
- El nuevo objeto es de aspecto feo.
- El nuevo objeto tiene algunas antenas. ¿Cómo son esas antenas?
    - Las antenas son de color violeta
    - Las antenas son de longitud extensa
- El nuevo objeto tiene un ojo. ¿Cómo es ese ojo?
    - El ojo es de forma oval
    - El ojo es de color azul
    - El ojo es de tamaño grande
- El nuevo objeto tiene unos cuantos pelos. ¿Cómo son esos pelos?
    - Los pelos son de color fucsia
    - Los pelos son de textura rugosa

(nuevas características)

- El nuevo objeto tiene un pie. ¿Cómo es ese

pie?

   - El pie es de forma rectangular
   - El pie es de color amarillo
   - El pie tiene 3 dedos. ¿Cómo son esos dedos?
      - Los dedos son de longitud mediana
      - Los dedos son de forma alargada
      - Los dedos son de color amarillo

Podemos observar como nuestro nuevo objeto es una especie de “objeto original ampliado”. Es decir que el nuevo objeto, es exactamente igual al objeto original (comparte todos sus atributos) pero posee nuevas características.

Está claro además, que el objeto original y el nuevo objeto, son dos objetos diferentes ¿cierto? No obstante, el nuevo objeto es un sub-tipo del objeto original.

Ahora sí, a complicarnos aún más.

Los objetos, también tienen la capacidad de “hacer cosas”

Ya describimos las cualidades de nuestros objetos. Pero de lo que no hemos hablado, es de aquellas cosas que los objetos "pueden hacer", es decir, "cuáles son sus capacidades".

Los objetos tiene la capacidad de realizar acciones. Las acciones, son verbos. Es decir, que para conocer las capacidades de un objeto, debes preguntarte "¿Qué puede hacer el objeto?" y la respuesta a esta pregunta, estará dada por todas aquellas que comiencen por la frase "el objeto puede" seguida de un verbo en infinitivo.

Algunos ejemplos:

- El objeto original puede flotar
- El nuevo objeto (además) puede saltar

Objetos y más objetos: la parte difícil

Si entendiste todo lo anterior, ahora viene la parte difícil. ¿Viste que esto de "pensando en objetos" viene a colación de la programación orientada a objetos? Bueno, la parte difícil es que en la programación, todo lo que acabamos de ver, se denomina de una forma particular. Pero, la

explicación es la misma que te di antes.

Al pan, pan. Y al vino, vino. Las cosas por su nombre

Cuando antes hablamos de...

- Objeto: en la POO[1], también se denomina **"objeto"**.
- Atributos y cualidades: en la POO se denominan **"propiedades"**.
- Acciones que puede realizar un objeto: en la POO, se denominan **"métodos"**.
- Atributos-objeto: en la POO se denomina **"composición"** y es una técnica.
- Objetos que tienen los mismos nombres de atributos (por ejemplo: *color, forma, etc.*): en la POO se denomina **"polimorfismo"**, y representa la capacidad que tienen los objetos, de poseer los mismos nombres de propiedades y métodos. El polimorfismo, es una característica esencial de la POO.
- Sub-tipos de objetos: en la POO se denomina

1 Programación orientada a objetos

“herencia”. Otra característica esencial de este paradigma.

Entendiendo el paradigma de la programación orientada a objetos

La Programación Orientada a Objetos (POO u OOP por sus siglas en inglés), es un paradigma de programación.

> *“Paradigma: teoría cuyo núcleo central [...] suministra la base y modelo para resolver problemas [...]”. Definición de la Real Academia Española, vigésimo tercera edición*

Cómo tal, nos enseña un método -probado y estudiado- el cual se basa en las interacciones de objetos (todo lo descrito en el título anterior, “Pensar en objetos”) para resolver las necesidades de un sistema informático.

Básicamente, este paradigma se compone de 6 elementos y 7 características que veremos a continuación.

Elementos y Características de la POO

Los elementos de la POO, pueden entenderse como los "materiales" que necesitamos para diseñar y programar un sistema, mientras que las características, podrían asumirse como las "herramientas" de las cuáles disponemos para construir el sistema con esos materiales.

Entre los elementos principales de la POO, podremos encontrar los siguientes:

Clases

Las clases son los modelos sobre los cuáles se construirán nuestros objetos.

Propiedades

Las propiedades, como hemos visto antes, son las características intrínsecas del objeto. Éstas, se representan a modo de variables, solo que técnicamente, pasan a denominarse "propiedades".

Métodos

Los métodos son "funciones" (como las que utilizamos en la programación estructurada), solo que técnicamente se denominan métodos, y representan acciones propias que puede realizar el

objeto (y no otro). Cada método debe tener una -y solo una- responsabilidad.

Objeto

Las clases por sí mismas, no son más que modelos que nos servirán para crear objetos en concreto. Podemos decir que una clase, es el razonamiento abstracto de un objeto, mientras que el objeto, es su materialización. A la acción de crear objetos, se la denomina "instanciar una clase" y dicha instancia, consiste en asignar la clase, como valor a una variable, la cual se convertirá en una "variable de tipo objeto", puesta que la misma, tendrá como valor, a las propiedades y métodos que hayan sido definidos en la clase.

Herencia

Como comentamos en el título anterior, algunos objetos comparten las mismas propiedades y métodos que otro objeto, y además agregan nuevas propiedades y métodos. A esto se lo denomina herencia: una clase que hereda de otra.

Composición

Como comentamos anteriormente, algunos objetos se componen de las propiedades de otro (lo cual,

no significa que las hereden, sino simplemente eso: "se componen de").

Cuando la propiedad de un objeto, se compone de las características de otro objeto, dicha propiedad se transforma en una especie de "propiedad-objeto". Es decir, que el tipo de datos de esta propiedad, pasa a ser de tipo objeto. Esto significa, que dicha propiedad, estará formada por sub-propiedades.

Capítulo II: Teoría sintáctico-gramatical de objetos y el diseño de objetos en lenguaje natural

La teoría sintáctico-gramatical de objetos, es un nuevo enfoque de la POO, el cual se construye sobre la base del lenguaje natural, para diseñar los objetos de un sistema informático, sus características y forma en la que se relacionan.

En los últimos cinco años, tras arduas horas de estudio y dedicación, pude comprobar que diseñando los objetos a partir del lenguaje natural, se logra definir y relacionar a los mismos, mediante conexiones lógicas irrefutables.

Siendo el idioma el lenguaje más evolucionado con el que la humanidad cuenta para comunicarse, hacer que los componentes de un sistema informático sean diseñados sobre la base de éste,

resulta la forma más óptima y segura de lograr una comunicación fehaciente, en el núcleo interno de un Software.

La teoría sintáctico-gramatical de objetos, propone definir los objetos -en una primera instancia-, pura y exclusivamente en lenguaje natural, aplicando para ello, dos estructuras gramaticales únicas, que deben ser respetadas de forma estricta, a fin de asegurar conexiones lógicas irrefutables.

A continuación, veremos en qué consiste y cómo implementar dicha técnica.

Diseñando un sistema informático en lenguaje natural

Diseñar un sistema informático orientado a objetos no es lo mismo que utilizar elementos de la orientación a objetos: *puedo pensar un sistema orientado a procedimientos y escribir el código como si estuviese orientado a objetos.* Lo anterior, *no es programar orientado a objetos*, sino, programar utilizando elementos de la POO.

El primer paso para diseñar un sistema orientado a objetos -según la teoría sintáctico-gramatical de

objetos-, es describir el objeto en lenguaje natural.

Para describir los objetos en lenguaje natural, los pasos a seguir son:

1. Comenzar por el objeto qué mayor cualidades presente
2. Describir SOLO una cualidad por vez, utilizando una de las siguientes estructuras gramaticales:
   - **Estructura gramatical adjetiva:** *La mesa es de material metálico.*
   - **Estructura gramatical sustantiva:** *La mesa tiene cuatro patas*.

Identificar Objetos: Es el primer sustantivo más los obtenidos de las estructuras gramaticales sustantivas: último sustantivo de una estructura gramatical sustantiva, en singular.

- La ***Mesa*** es de color blanco.
- La mesa tiene cuatro ***patas***. (el objeto será en singular: **Pata**)

  - La pata es de color blanco.

Identificar propiedades:

PROPIEDAD SIMPLE: En una estructura gramatical adjetiva, es el sustantivo que forma una locución adjetiva (pertenece al objeto que le antecede).

- La mesa *es de color* blanco. (***color*** es propiedad de mesa)
- La mesa tiene cuatro patas.
  - La pata *es de color* blanco. (***color*** es propiedad de pata)

PROPIEDAD COMPUESTA: El último sustantivo de una estructura gramatical sustantiva (es propiedad del objeto que le antecede).

- La mesa es de color blanco.
- La mesa tiene cuatro **patas**. (es propiedad compuesta de mesa)
  - La pata es de color blanco.

Si la definición de los objetos, se realiza empleando

un sistema visual de niveles (por ejemplo, las viñetas con niveles), será sumamente sencillo, reconocer los objetos, sus propiedades, relación y dependencias de los mismos.

Definamos una habitación con cuatro paredes cubiertas de lajas, una puerta y una ventana:

- La habitación tiene cuatro paredes

(defino todas las características de las paredes, refiriéndome solo a una)

    - La pared es de posición lateral
    - La pared tiene muchas lajas
        - La laja es de textura rugosa
        - La laja es de color beige
        - La laja es de longitud 250 mm
        - La laja es de anchura 150 mm
    - La pared tiene una puerta
        - La puerta es de material madera
        - La puerta es de color cedro

    - La pared tiene una ventana
        - La ventana tiene un marco
            - El marco es de material aluminio
            - El marco es de color blanco
            - El marco tiene un vidrio
                - El vidrio es de espesor 10 mm
                - El vidrio es de textura lisa
                - El vidrio es de pigmentación blancuzca
- La habitación tiene un cielo raso[2]
    - El cielo raso es de material yeso
    - El cielo raso es de color blanco
- La habitación tiene un piso

2 Cuando solía definir las propiedades de una habitación, me refería al cielo raso como “techo”. El 15 de septiembre de 2012, mientras daba un taller de orientación a objetos para NeaExtends (en la provincia de Formosa, Argentina), uno de los asistentes, muy sabiamente me hizo notar que el techo, es la parte del exterior de la habitación, mientras que lo que se puede visualizar desde el interior, se denomina “cielo raso”.

    - El piso es de tipo flotante
    - El piso es de material madera
    - El piso es de color cedro

Una vez realizada la descripción de los objetos en lenguaje natural, se procede a la identificación de objetos y propiedades:

- La **habitación** tiene cuatro _paredes_
    - La **pared** es de _posición_ lateral
    - La pared tiene muchas _lajas_
        - La **laja** es de _textura_ rugosa
        - La laja es de _color_ beige
        - La laja es de _longitud_ 250 mm
        - La laja es de _anchura_ 150 mm
    - La pared tiene una _puerta_
        - La **puerta** es de _material_ madera
        - La puerta es de _color_ cedro

    - La pared tiene una *ventana*
        - La **ventana** tiene un *marco*
            - El **marco** es de *material* aluminio
            - El marco es de *color* blanco
            - El marco tiene un *vidrio*
                - El **vidrio** es de *espesor* 10 mm
                - El vidrio es de *textura* lisa
                - El vidrio es de *pigmentación* blancuzca
- La habitación tiene un *cielo raso*
    - El **cielo raso** es de *material* yeso
    - El cielo raso es de *color* blanco
- La habitación tiene un *piso*
    - El **piso** es de *tipo* flotante
    - El piso es de *material* madera
    - El piso es de *color* cedro

Notar que hemos resaltado los **objetos** solo una vez, mientras que las *propiedades*, han sido todas subrayadas y colocadas en cursiva.

Del lenguaje natural al lenguaje de programación

Una vez definidos en lenguaje natural todos nuestros objetos, estamos en condiciones de transcribirlos en un lenguaje que pueda ser interpretado por el ordenador.

Para ello, basaremos todos los ejemplos en Python[3] y PHP[4].

Definición de clases

Al pasar del lenguaje natural al lenguaje de programación, siempre comenzaremos por aquellos objetos que menor cantidad de dependencias posean.

Python:
```
class Vidrio(object):
    pass
```

PHP:
```
class Vidrio {
}
```

Definición de propiedades

La definición de propiedades se hará directamente

3 http://www.python.org

4 http://www.php.net

en el método constructor[5] de la clase.

Dado que Python y PHP son dos lenguajes de tipado dinámico[6], en el método constructor se asignará a cada propiedad, un valor equivalente a cero, representativo del tipo de datos.

Se considerarán valores equivalentes a cero:

```
Equivalente a cero de tipo entero:   0
Equivalente a cero de tipo real:     0.0
Equivalente a cero de tipo string:   ''
Equivalente a cero de tipo booleano: False
Equivalente a cero de tipo colección:
(en propiedades compuestas por más de un objeto)
                                 PHP: array()
                              Python: []
```

Cuando una propiedad se componga de un solo objeto, se asignará como valor de la misma, una instancia de dicho objeto:

En PHP: `new Objeto()`   **En Python:** `Objeto()`

---

5 El método constructor es aquel que se ejecuta de forma automática para inicializar un objeto cuando éste es creado. El nombre del método constructor se encuentra reservado, siendo __init__() para Python y __construct() para PHP.

6 Tipado dinámico: dícese de aquellos lenguajes de programación generalmente interpretados, en los cuales una misma variable puede tomar valores de distinto tipo a lo largo de la ejecución del programa. Dichos programas, no requieren que a una variable se le defina de forma anticipada su tipo de datos y por lo tanto, no existen restricciones al respecto.

Un ejemplo concreto:

Python:
```
class UnObjeto(object):

    def __init__(self):
        self.entero = 0
        self.real = 0.0
        self.cadena = ''
        self.booleano = False
        self.objetos = []
        self.objeto = Objeto()
```

PHP:
```
class UnObjeto {

    function __construct() {
        $this->entero = 0;
        $this->real = 0.0;
        $this->cadena = '';
        $this->booleano = False;
        $this->objetos = array();
        $this->objeto = new Objeto();
    }

}
```

*Asignar a una propiedad la instancia de una clase, se denomina **"composición"**.*

En los objetos, nos podremos encontrar tres tipos de propiedades:

1. **Propiedades simples** (de tipo único)
2. **Propiedades compuestas** (compuestas por un solo objeto)

3. **Propiedades colectoras** (compuestas por una colección de objetos)

Capítulo III: Creación de objetos y acceso a propiedades

Anteriormente, comentamos que la creación de un objeto consiste en asignar a una variable, la instancia de una clase. Es decir, que con la definición de los objetos, solo contamos con un "molde" para "cocinar" nuestros objetos, pero aún, no contamos con el objeto propiamente dicho.

Crear un objeto es sumamente simple:

Python:

```
objeto = Objeto()
```

PHP:

```
$objeto = new Objeto();
```

Una vez creado nuestro objeto, podremos ver su estructura interna:

Python[7]:

```
from printr import printr
printr(objeto)
```

7 Para ver la estructura interna de un objeto en Python, se recomienda descargar el módulo python-printr de www.python-printr.org e instalarlo en /usr/lib/Python2.x/

PHP:
```
print_r($objeto);
```

Veamos un ejemplo completo con vidrio, marco y ventana, para así, poder modificar el valor de sus propiedades:

Python:

```
class Vidrio(object):

    def __init__(self):
        self.espesor = 0
        self.textura = ''
        self.pigmentacion = ''


class Marco(object):

    def __init__(self):
        self.material = ''
        self.color = ''
        self.vidrio = Vidrio()


class Ventana(object):

    def __init__(self):
        self.marco = Marco()


ventana = Ventana()

from printr import printr
printr(ventana)
```

El código anterior, arrojará la siguiente salida::

```
<Ventana object>
    {
        marco: <Marco object>
        {
            color: ''
            vidrio: <Vidrio object>
            {
                espesor: 0
                textura: ''
                pigmentacion: ''
            }
            material: ''
        }
    }
```

Mientras que el mismo ejemplo en PHP:

PHP:

```
class Vidrio{

    function __construct() {
        $this->espesor = 0;
        $this->textura = '';
        $this->pigmentacion = '';
    }

}

class Marco {

    function __construct() {
        $this->material = '';
        $this->color = '';
        $this->vidrio = new Vidrio();
    }

}
```

```
class Ventana {

    function __construct() {
        $this->marco = new Marco();
    }

}

$ventana = new Ventana();
print_r($ventana);
```

Lo arrojará de la siguiente forma:

```
Ventana Object
(
    [marco] => Marco Object
        (
            [material] =>
            [color] =>
            [vidrio] => Vidrio Object
                (
                    [espesor] => 0
                    [textura] =>
                    [pigmentacion] =>
                )

        )

)
```

El objeto fue creado (inicializado). Sin embargo, ningún valor ha sido asociado a sus propiedades. Si las propiedades fuesen simples y quisiéramos modificarlas, lo haríamos de la siguiente forma:

Python:
```
objeto.propiedad = 'nuevo valor'
```

PHP:
```
$objeto->propiedad = 'nuevo valor';
```

Pero cuando las propiedades son compuestas, debe seguirse toda "la ruta" de propiedades, hasta llegar a la deseada:

Python:
```
ventana.marco.vidrio.espesor = 10
```

PHP:
```
$ventana->marco->vidrio->espesor = 10;
```

En objetos donde la dependencia es de varios niveles, modificar las propiedades de esta forma, no solo se hace difícil, sino que además, es bastante confuso.

Claro que también podríamos crear un nuevo objeto, modificarle sus propiedades simples y reasignarlo al objeto compuesto:

Python:
```
vidrio = Vidrio()
vidrio.espesor = 10

marco = Marco()
marco.vidrio = vidrio

ventana = Ventana()
ventana.marco = marco
```

```
PHP:
$vidrio = new Vidrio();
$vidrio->espesor = 10;

$marco = new Marco();
$marco->vidrio = $vidrio;

$ventana = new Ventana();
$ventana->marco = $marco;
```

Pero haciéndolo de esa forma, también podríamos confundirnos y modificar -sin quererlo- la estructura interna del objeto:

```
Python:
vidrio = Vidrio()
vidrio.espesor = 10

ventana = Ventana()
ventana.marco = vidrio

PHP:
$vidrio = new Vidrio();
$vidrio->espesor = 10;

$ventana = new Ventana();
$ventana->marco = $vidrio;
```

Es entonces, cuando se hace necesario en grandes relaciones de objetos, contar con una solución a este problema. Y la misma, es provista por el patrón de diseño compuesto que veremos en el siguiente capítulo.

Capítulo IV: Patrón de diseño compuesto (composite pattern)

Antes de hablar de patrones de diseño, es necesario familiarizarse con el estudio de la Arquitectura de Software.

A continuación, se realizará una breve introducción, a fin de alcanzar el conocimiento mínimo necesario para poder afrontar finalmente, el estudio de este patrón de diseño compuesto.

Introducción a la Arquitectura de Software

¿Qué es la arquitectura de software?

Es necesario aclarar, que no existe una definición única, exacta, abarcadora e inequívoca de "arquitectura de software". La bibliografía sobre el tema es tan extensa como la cantidad de definiciones que en ella se puede encontrar. Por lo tanto trataré, no de definir la arquitectura de software, sino más bien, de introducir a un

concepto simple y sencillo que permita comprender el punto de vista desde el cual, este libro abarca a la arquitectura de software pero, sin ánimo de que ello represente "una definición más".

A grandes rasgos, puede decirse que "la Arquitectura de Software es la forma en la que se organizan los componentes de un sistema, interactúan y se relacionan entre sí y con el contexto, aplicando normas y principios de diseño y calidad, que fortalezcan y fomenten la *usabilidad* a la vez que dejan preparado el sistema, para su propia evolución".

Atributos de calidad

La Calidad del Software puede definirse como los atributos implícitamente requeridos en un sistema que deben ser satisfechos. Cuando estos atributos son satisfechos, puede decirse (aunque en forma objetable), que la calidad del software es satisfactoria. Estos atributos, se gestan desde la arquitectura de software que se emplea, ya sea cumpliendo con aquellos requeridos durante la ejecución del software, como con aquellos que forman parte del proceso de desarrollo de éste.

Atributos de calidad que pueden observarse durante la ejecución del software

1. Disponibilidad de uso

2. Confidencialidad, puesto que se debe evitar el acceso no autorizado al sistema

3. Cumplimiento de la Funcionalidad requerida

4. Desempeño del sistema con respecto a factores tales como la capacidad de respuesta

5. Confiabilidad dada por la constancia operativa y permanente del sistema

6. Seguridad externa evitando la pérdida de información debido a errores del sistema

7. Seguridad interna siendo capaz de impedir ataques, usos no autorizados, etc.

Atributos de calidad inherentes al proceso de desarrollo del software

1. Capacidad de Configuración que el sistema otorga al usuario a fin de realizar ciertos cambios

2. Integrabilidad de los módulos independientes del sistema

3. Integridad de la información asociada

4. Capacidad de Interoperar con otros sistemas (interoperabilidad)

5. Capacidad de permitir ser modificable a futuro (modificabilidad)

6. Ser fácilmente Mantenible (mantenibilidad)

7. Capacidad de Portabilidad, es decir que pueda ser ejecutado en diversos ambientes tanto de software como de hardware

8. Tener una estructura que facilite la Reusabilidad de la misma en futuros sistemas

9. Mantener un diseño arquitectónico Escalable que permita su ampliación (escalabilidad)

10. Facilidad de ser Sometido a Pruebas que aseguren que el sistema falla cuando es lo que se espera (*testeabilidad*)

Niveles de abstracción

Podemos decir que la AS[8] se divide en tres niveles de abstracción bien diferenciados: **Estilo Arquitectónico**, **Patrón Arquitectónico** y **Patrón de Diseño**. Existe una diferencia radical entre estos tres elementos, que debe marcarse a fin de evitar las grandes confusiones que inevitablemente,

8 Arquitectura de Software

concluyen en el mal entendimiento y en los resultados poco satisfactorios. Éstos, son los que en definitiva, aportarán "calidad" al sistema resultante. En lo sucesivo, trataremos de establecer la diferencia entre estos tres conceptos, viendo como los mismos, se relacionan entre sí, formando parte de un todo: la arquitectura de software.

Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño, representan -de lo general a lo particular- los tres niveles de abstracción en los que se divide la Arquitectura de Software.

Estilo Arquitectónico

El estilo arquitectónico define un nivel general de la estructura del sistema y cómo éste, va a comportarse. Mary Shaw y David Garlan, en su libro "Software Architecture" (Prentice Hall, 1996), definen los estilos arquitectónicos como la forma de determinar el los componentes y conectores de un sistema, que pueden ser utilizados a instancias del estilo elegido, conjuntamente con un grupo de restricciones sobre como éstos pueden ser combinados:

> *"[...] an architectural style determines the vocabulary of components and connectors that*

can be used in instances of that style, together with a set of constraints on how they can be combined [...]"

Mary Shaw y David Garlan -en el mismo libro-, hacen una distinción de estilos arquitectónicos comunes, citando como tales a:

1. Pipes and filters (filtros y tuberías)
2. Data Abstraction and Object-Oriented Organization (Abstracción de datos y organización orientada a objetos)
3. Event-based (estilo basado en eventos)
4. Layered Systems (Sistemas en capas)
5. Repositories (Repositorios)
6. Table Driven Interpreters

Viendo la clasificación anterior, es muy frecuente que se encuentren relaciones entre los estilos arquitectónicos y los paradigmas de programación. Sin embargo, debe evitarse relacionarlos en forma directa.

Patrón Arquitectónico

Un patrón arquitectónico, definirá entonces, una

plantilla para construir el Software, siendo una particularidad del estilo arquitectónico elegido.

En esta definición, es donde se incluye a MVC, patrón que a la vez, puede ser enmarcado dentro del estilo arquitectónico orientado a objetos (estilo arquitectónico basado en el paradigma de programación orientada a objetos).

Patrón de Diseño

Dentro de niveles de abstracción de la arquitectura de Software, los patrones de diseño representan el nivel de abstracción más detallado. A nivel general, nos encontramos con el Estilo Arquitectónico. En lo particular, hallamos al Patrón Arquitectónico y, finalmente, el Patrón de Diseño es “el detalle”.

Matt Zandstra en su libro “PHP Objects, Patterns and Practice” (Apress, 2010) define los patrones de diseño como:

“[…] is a problem analyzed with good practice for its solution explained [...]”

(Traducción: un problema analizado con buenas prácticas para su solución explicada)

Un patrón de diseño, entonces, es un análisis

mucho más detallado, preciso y minucioso de una parte más pequeña del sistema, que puede incluso, trabajar en interacción con otros patrones de diseño. Por ejemplo, un Singleton puede coexistir con un Factory y éstos, a la vez, con Composite.

En este sentido, un Patrón Arquitectónico como MVC, podría utilizar diversos patrones de diseño en perfecta coexistencia, para la creación de sus componentes.

Composite design pattern (patrón de diseño compuesto)

Habiendo comprendido de qué hablamos al referirnos a un patrón de diseño, vamos a intentar implementar la solución a nuestro anterior problema, mediante el patrón de diseño compuesto.

El patrón de diseño compuesto, nos permite componer una propiedad asignándole como valor, al objeto compositor[9] de forma directa (objeto que

---

9 En una gran parte de la bibliografía sobre programación orientada a objetos y patrones de diseño, encontrarás que al objeto compositor se lo define como "componente". Definir a un objeto cuya responsabilidad es componer a otro, como "componente", es un grave error, puesto que los "componentes" de un sistema informático, son aquellos elementos que lo

compone a dicha propiedad), previniendo que por error, le sea asignado un objeto o valor, diferente al tipo del objeto esperado. Para lograrlo, dicho patrón esperará que el objeto compositor, le sea pasado como parámetro.

Composite en PHP

En PHP 5.x, es posible indicar a un método, el tipo de objeto esperado, anteponiendo el nombre del mismo:

```
class Marco {

    function __construct(Vidrio $vidrio=NULL) {
        $this->material = '';
        $this->color = '';
        $this->vidrio = $vidrio;
    }

}
```

De esta forma, si intentáramos construir el objeto Marco(), pasando como parámetro un objeto que

integran y dicho sistema, no necesariamente estará orientado a objetos. Para la teoría sintáctico-gramatical de objetos, el idioma español cumple un papel fundamental. La Real Academia Española, define el término “compositor” como un adjetivo cuyo significado es “que compone”. Y en este sentido será utilizado en este libro, a fin de no confundirlo con “los componentes de un sistema informático”.

no sea de tipo Vidrio(), un error será lanzado:

```
$marco = new Marco('no soy un vidrio');

PHP Catchable fatal error:  Argument 1 passed to
Marco::__construct() must be an instance of Vidrio,
string given
```

Sin embargo, si un objeto Vidrio() o NULL le es pasado como parámetro, el patrón habrá servido a su fin:

```
$vidrio = new Vidrio();
$vidrio->espesor = 10;
$vidrio->textura = 'lisa';
$vidrio->pigmentacion = 'blancuzca';

$marco = new Marco($vidrio);
print_r($marco);

Marco Object
(
    [material] =>
    [color] =>
    [vidrio] => Vidrio Object
        (
            [espesor] => 10
            [textura] => lisa
            [pigmentacion] => blancuzca
        )

)
```

Composite en Python

A diferencia de PHP, en Python no podemos indicar a un método, el tipo de objeto que estamos

esperando. Para componer mediante *«composite»* en Python, la función isinstance(objeto, clase)[10] será necesaria:

```
class Marco(object):

    def __init__(self, vidrio=None):
        self.material = ''
        self.color = ''
        if isinstance(vidrio, Vidrio) or vidrio is None:
            self.vidrio = vidrio
        else:
            raise TypeError('%s no es de tipo Vidrio'
                % type(vidrio))
```

Si un objeto de tipo diferente a Vidrio le es pasado como parámetro, un error será lanzado:

```
marco = Marco('no soy un vidrio')
TypeError: <type 'str'> no es de tipo Vidrio
```

De lo contrario, se ejecutará satisfactoriamente:

```
vidrio = Vidrio()
vidrio.espesor = 10
vidrio.textura = 'lisa'
vidrio.pigmentacion = 'blancuzca'

marco = Marco(vidrio)
```

En el caso de Python, el patrón compuesto “nos

10 http://docs.python.org/library/functions.html#isinstance

salva" al tiempo que, implementado de la forma anterior, "nos ensucia" el código además de obligarnos a caer en redundancia.

Para solucionar este problema, podemos generar nuestra propia función *compose()* y así, mantener "limpios" nuestros constructores y evitar la redundancia en un solo paso:

```
def compose(obj, cls):
    if isinstance(obj, cls) or obj is None:
        return obj
    else:
        raise TypeError('%s no es de tipo %s'
                % (type(obj), cls))

class Marco(object):

    def __init__(self, vidrio=None):
        self.material = ''
        self.color = ''
        self.vidrio = compose(vidrio, Vidrio)
```

Capítulo V: Objetos compositores exclusivos, identidad, pertenencia y agregación

Identidad

Todos los objetos poseen una identidad que los hace únicos. Cuando se crea un objeto, éste, adopta una identidad que será diferente a la de otro objeto, incluso cuando sus cualidades sean exactamente las mismas. Es decir, yo puedo crear dos objetos cuyas cualidades sean exactamente idénticas y, sin embargo, serán dos objetos distintos.

De la misma forma que dos personas gemelas, no solo son exactamente iguales a la vista, sino que además, comparten hasta el mismo ADN, dos objetos (o más) que cuenten con el mismo "ADN", a pesar de su igualdad, deben poder ser diferenciados "más allá de la vista" y contar con una identidad.

Dos hermanos gemelos se diferenciarán entre sí por

su número de documento (DNI, Pasaporte, etc.), mientras que el DNI de nuestros objetos, será su ID.

Por lo tanto, a toda definición de clase, siempre se le agregará en el método constructor -y como primera propiedad- la propiedad objeto_id, donde “objeto” será el nombre de la clase:

Python:
```
class Vidrio(object):

    def __init__(self):
        self.vidrio_id = 0
        self.espesor = 0
        self.textura = ''
        self.pigmentacion = ''
```

PHP:
```
class Vidrio{

    function __construct() {
        $this->vidrio_id = 0;
        $this->espesor = 0;
        $this->textura = '';
        $this->pigmentacion = '';
    }

}
```

Objetos compositores

Todo objeto que sirve para componer a otro, se denomina objeto compositor.

Entre los objetos compositores, podemos encontrar dos tipos claramente diferenciados: los compositores reutilizables y los compositores exclusivos -o de pertenencia-, en los que nos centraremos en este capítulo.

Objetos compositores exclusivos

Se dice que un compositor es exclusivo cuando éste puede componer únicamente a un -y solo un- objeto.

Cuando la propiedad de un objeto se compone de *N* objetos del mismo tipo pero diferente identidad (es decir, se compone de objetos similares pero no idénticos), dichos compositores serán «compositores exclusivos».

Para que un compositor sea considerado de pertenencia, deben necesitarse al menos dos compositores para componer la misma propiedad.

Por ejemplo, puedo decir que un vidrio compone a un marco. Pero solo es 1 vidrio. Por lo tanto, no es compositor de pertenencia. También puedo decir que 75 lajas componen a la misma pared. Sin embargo, puedo crear una única laja y replicar la misma laja *N* veces para componer a esa pared y a otras. Es decir, puedo reutilizar una misma laja

para componer más de una pared. Entonces, hablo de un compositor reutilizable. Pero cuando digo que 4 paredes componen a la misma habitación, cada una de ellas se diferencia por sus cualidades y no puedo reutilizar la misma pared para componer más de una habitación. De hecho ¿podrías trasladar la pared del living al dormitorio? Es aquí cuando hablamos de objetos compositores de pertenencia: *«El Objeto A pertenece al Objeto B»*.

El caso de Pared y Habitación, es exactamente el mismo que se da -por ejemplo- en un sistema de administración de empleados, donde:

«El empleado tiene varios datos de contacto»

«El dato de contacto es de tipo teléfono móvil»

y finalmente, podemos decir que:

«El dato de contacto pertenece al empleado».

Pertenencia

Cuando un tipo de objeto es considerado un compositor exclusivo, dicha característica se define con una nueva

estructura gramatical:

El [objeto A] pertenece [a|al|a la] [objeto B]

«La pared pertenece a la habitación»

Dicha estructura gramatical, deriva en una nueva propiedad del objeto compositor: la propiedad objeto_compuesto, cuyo valor, será la identidad del objeto al cuál compone:

Python:
```
class Pared(object):

    def __init__(self, puerta=None, ventana=None):
        self.pared_id = 0
        self.lajas = []
        self.puerta = compose(puerta , Puerta)
        self.ventana = compose(ventana , Ventana)
        self.habitacion = 0
```

PHP:
```
class Pared {

    function __construct(Puerta $puerta=NULL,
                         Ventana $ventana=NULL) {
        $this->pared_id = 0;
        $this->lajas = array();
        $this->puerta = $puerta;
        $this->ventana = $ventana;
        $this->habitacion = 0;
    }

}
```

Agregación

Cuando una propiedad se compone por varios objetos del mismo tipo, la misma se define como una colección. Ya sea un array en PHP como una lista en Python.

Al crear el objeto, se le deben ir "agregando" tantos compositores como sean necesarios para componer dicha propiedad:

Python:
```
pared_izquierda = Pared()
pared_derecha = Pared()
pared_frontal = Pared()
pared_dorsal = Pared()

habitacion = Habitacion(piso, cielo_raso)
habitacion.paredes.append(pared_izquierda)
habitacion.paredes.append(pared_derecha)
habitacion.paredes.append(pared_frontal)
habitacion.paredes.append(pared_dorsal)
```

PHP:
```
$pared_izquierda = new Pared();
$pared_derecha = new Pared();
$pared_frontal = new Pared();
$pared_dorsal = new Pared();

$habitacion = new Habitacion($piso, $cielo_raso);
$habitacion->paredes[] = $pared_izquierda;
$habitacion->paredes[] = $pared_derecha;
$habitacion->paredes[] = $pared_frontal;
$habitacion->paredes[] = $pared_dorsal;
```

Al igual que nos pasaba al inicializar un objeto, podríamos agregar un objeto del tipo incorrecto:

Python:
```
habitacion = Habitacion(piso, cielo_raso)
habitacion.paredes.append('no soy una pared')
```

PHP:
```
$habitacion = new Habitacion($piso, $cielo_raso);
$habitacion->paredes[] = 'no soy una pared';
```

Y nuestro sistema, seguiría adelante como si nada malo hubiese sucedido, poniendo en riesgo toda la integridad de la aplicación.

Para resolver este problema, sabemos que existe el patrón compuesto pero ¿cómo implementarlo? Para ello, se utilizan los llamados «métodos de agregación».

Los métodos de agregación, no son más que funciones que utilizando el patrón compuesto, se encargan de "agregar" los compositores que sean necesarios para componer la propiedad colectora. Dichos métodos, se encuentran presentes en todo objeto que posea propiedades colectoras.

Vale aclarar que los métodos de agregación, no deben definir un valor nulo por defecto como sucede en los métodos constructores cuando se utiliza el patrón compuesto.

Python:
```
class Habitacion(object):

    def __init__(self, piso, cr) :
        self.habitacion_id = 0
        self.paredes = []
        self.piso = compose(piso, Piso)
        self.cielo_raso = compose(cr, CieloRaso)


    def add_pared(self, pared) :
        self.paredes.append(compose(pared, Pared))
```

PHP:
```
class Habitacion {

    function __construct(Piso $piso=NULL,
                         CieloRaso $cr=NULL) {
        $this->habitacion_id = 0;
        $this->paredes = array();
        $this->piso = $piso;
        $this->cielo_raso = $cr;
    }

    function add_pared(Pared $pared) {
        $this->paredes[] = $pared;
    }

}
```

Con los métodos de agregación, el agregado de compositores resultará mucho más seguro y sustentable:

Python:
```
pared_izquierda = Pared()
pared_derecha = Pared()
pared_frontal = Pared()
pared_dorsal = Pared()
```

```
habitacion = Habitacion(piso, cielo_raso)
habitacion.add_pared(pared_izquierda)
habitacion.add_pared(pared_derecha)
habitacion.add_pared('no soy pared')  # Fallará
habitacion.add_pared(pared_dorsal)
```

PHP:

```
$pared_izquierda = new Pared();
$pared_derecha = new Pared();
$pared_frontal = new Pared();
$pared_dorsal = new Pared();

$habitacion = new Habitacion($piso, $cielo_raso);
$habitacion->add_pared($pared_izquierda);
$habitacion->add_pared($pared_derecha);
$habitacion->add_pared('no soy pared');  # Fallará
$habitacion->add_pared($pared_dorsal);
```

Capítulo VI: Objetos relacionales simples -o multiplicadores-

Imagina que tienes que agregar 75 lajas a una pared. Y que en realidad, solo tienes una laja y nada más, deseas replicarla 75 veces para componer la misma pared.

Una pared, se compone de 75 lajas idénticas (las lajas son exactamente iguales entre sí). Es decir, que con tan solo crear 1 objeto laja podremos replicarlo N veces -sin necesidad de modificarlo-, para componer esa u otras paredes. Hablamos de 75 lajas con la misma identidad. Podemos decir entonces, que ***el objeto laja, es un objeto compositor reutilizable****.*

Claro está que podrías colocar un bucle en el método de agregación add_laja() del objeto Pared. Pero al no ser laja un compositor exclusivo ¿cómo sabrás la identidad de la laja que compone a una determinada pared? Y sin conocer la identidad de

la laja que se necesita para componer una pared ¿cómo harías para recuperarla tras haberla creado?

Para resolver este problema, es entonces, que surge la necesidad de contar con un objeto relacional multiplicador: para poder establecer la relación existente entre 1 objeto compuesto y *N* objetos compositores reutilizables (objetos idénticos -misma identidad-).

Haciendo un paralelismo con el ejemplo de empleados y datos de contactos que vimos en el capítulo anterior, podemos decir que el caso de Laja y Pared será el mismo que acontezca -por ejemplo- en un sistema de gestión contable donde *«La nota de pedido tiene varios productos»*. Un producto podría ser "Paquete de Algodón Suavecito-Suavecito x 200 gr" y la misma nota de pedido, tener 40 paquetes del mismo producto.

Objeto relacional simple (o multiplicador)

Un objeto relacional es aquel cuya única finalidad es la de establecer la relación existente entre dos objetos, multiplicando *N* veces al compositor.

Un objeto relacional simple, se conforma de cuatro

propiedades:

- *INT* objeto_id (presente en todos los objetos)
- *ObjetoCompositor* objeto_compositor
- *ObjetoCompuesto* objeto_compuesto
- *INT* relacion

Para establecer la relación compositor/compuesto, incorporará un método relacional set_relacion() que de forma iterativa, agregará los compositores necesarios, recurriendo al método de agregación del objeto compuesto:

Python:
```
class LajaPared(object):

    def __init__(self, compuesto,
        compositor=None):
        self.lajapared_id = 0
        self.pared = compose(compuesto, Pared)
        self.laja = compose(compositor, Laja)
        self.relacion = 1

    def set_relacion(self):
        tmpvar = 0
        while tmpvar < self.relacion:
            self.pared.add_laja(self.laja)
            tmpvar += 1
```

PHP:
```
class LajaPared {

    function __construct(Pared $compuesto,
```

```
                        Laja $compositor=NULL) {

        $this->lajapared_id = 0;
        $this->pared = $compuesto;
        $this->laja = $compositor;
        $this->relacion = 1;
    }

    function set_relacion() {
        $tmpvar = 0;
        while($tmpvar < $this->relacion) {
            $this->pared->add_laja($this->laja);
            $tmpvar++;
        }
    }

}
```

Para establecer la relación entre N lajas y pared, solo bastará con:

1. Crear el objeto compositor
2. Crear el objeto compuesto
3. Crear el objeto relacional
4. Invocar al método set_relacion() para establecer la relación entre compositor * *N* y compuesto.

Python:

```
laja = Laja()
laja.color = 'beige'

pared = Pared(puerta, ventana)
pared.posicion = 'frontal'
```

```
relacional = LajaPared(pared, laja)
relacional.relacion = 75
relacional.set_relacion()
```

PHP:

```
$laja = new Laja();
$laja->color = 'beige';

$pared = new Pared($puerta, $ventana);
$pared->posicion = 'frontal';

$relacional = new LajaPared($pared, $laja);
$relacional.relacion = 75
$relacional.set_relacion()
```

Capítulo VII: Mapeo relacional de objetos y estructuras de almacenamiento de datos sustentables

Solo una vez que los objetos hayan sido definidos, serán mapeados relacionalmente -de forma manual-, para así obtener el diseño de la base de datos con una estructura de almacenamiento sustentable.

En la orientación a objetos, la base de datos debe cumplir una función meramente complementaria, cuyo objetivo, solo será servir de soporte para almacenar datos de forma sustentable que nos permitan recuperar -con cualquier propósito- objetos creados.

El objetivo de mapear objetos relacionalmente es "recorrer los objetos analizando su relación, a fin de poder diseñar una estructura de almacenamiento

de datos (base de datos) sustentable al servicio de dichos objetos".

¿Para qué mapear los objetos? Cuando uno crea objetos, los mismos son "volátiles". Es decir, se crea un objeto, se ejecuta el archivo y tras la finalización del *script*, el objeto se destruye y es eliminado de la memoria.

Entonces: ¿cómo recuperar luego ese objeto? Almacenar sus datos en una base de datos es la solución más práctica y viable. Por ese motivo, es que la necesitamos.

El mapeo relacional debe comenzarse por los objetos que menor cantidad de dependencias posean. Comenzando entonces, por aquellos objetos sin dependencias, seguiremos en el orden de menor a mayor cantidad de dependencias.

Para escribir las consultas SQL destinadas a crear las tablas de la base de datos, se deberán tener en cuenta las siguientes pautas:

Todos los objetos deben tener una tabla homónima

Más adelante, veremos que los únicos objetos que no tendrán una tabla homónima, serán los Singleton colectores y los subtipos simples.

El nombre de la tabla, será el nombre de la clase, en minúsculas.

```
Class MiClase
```

Producirá:

```
CREATE TABLE miclase
```

Todas las tablas deben crearse con el motor InnoDB

Esto es, debido a que nuestra base de datos, deberá contemplar las relaciones derivadas del diseño de los objetos. Por tanto, la base de datos, será una base de datos relacional.

MySQL utiliza por defecto el motor MyISAM, quien no posee soporte para bases de datos relaciones. En cambio, el motor InnoDB, nos provee de un excelente soporte para la manipulación de claves foráneas (foreign keys o FK).

```
CREATE TABLE miclase (
) ENGINE=InnoDB;
```

La propiedad objeto_id será clave primaria de la tabla

Una clave primaria, será de tipo entero, auto-incremental y no podrá permitir valores nulos.

```
CREATE TABLE miclase (
  miclase_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY KEY
) ENGINE=InnoDB;
```

Todas las propiedades simples, siempre serán campos de la tabla

Es decir, cualquier propiedad cuyo valor no sea una colección, ni otro objeto, será campo de la tabla, respetando el tipo de datos definido en el constructor.

Los campos de la tabla deberán tener sí o sí, el mismo nombre que las propiedades[11], sin ningún tipo de agregado o cambio.

---

11 Siempre digo que "enseñando también se aprende" y que "en vez de competir hay que saber aprender de otros". Si miras los estilos de escritura del código SQL, notarás que las comas (,) las coloco a la izquierda de los campos y no al final. Esto lo aprendí de *"Magoo"*, un alumno del curso de Orientación a Objetos en Python. Su técnica me resultó sorprenderte ya que al colocar las comas a la izquierda del campo, es prácticamente imposible olvidarse una.

```
CREATE TABLE miclase (
    miclase_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY KEY
  , propiedad_entero INT(3)
  , propiedad_real DECIMAL(6, 2)
  , propiedad_string VARCHAR(30)
  , propiedad_bool BOOL
) ENGINE=InnoDB;
```

A fin de lograr una estructura de datos optimizada, se deberá tener especial cuidado en la cantidad de dígitos o caracteres a indicar en los campos de tipo INT, DECIMAL, CHAR y VARCHAR.

Las propiedades de pertenencia, serán claves foráneas con efecto en cascada

Las propiedades de pertenencia, serán siempre de tipo INT(11). No podrán ser nulas y cuando la clave primaria a la que hagan referencia sea eliminada, producirán un efecto en cascada.

```
CREATE TABLE miclase (
    miclase_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY KEY
  , propiedad_entero INT(3)
  , propiedad_real DECIMAL(6, 2)
  , propiedad_string VARCHAR(30)
  , propiedad_bool BOOL
```

```
  , propiedad_de_pertenencia INT(11) NOT NULL
  , FOREIGN KEY (propiedad_de_pertenencia)
        REFERENCES tabla (campo_id)
        ON DELETE CASCADE
) ENGINE=InnoDB;
```

Las propiedades compuestas por un solo objeto serán campos de la tabla

Deberán ser de tipo entero admitiendo valores nulos. Serán establecidas como claves foráneas produciendo un valor nulo cuando la clave primaria sea eliminada, excepto cuando se trate de objetos relacionales, donde el efecto será en cascada.

```
CREATE TABLE miclase (
    miclase_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY KEY
  , propiedad_entero INT(3)
  , propiedad_real DECIMAL(6, 2)
  , propiedad_string VARCHAR(30)
  , propiedad_bool BOOL
  , propiedad_de_pertenencia INT(11) NOT NULL
  , FOREIGN KEY (propiedad_de_pertenencia)
        REFERENCES tabla (campo_id)
        ON DELETE CASCADE
  , propiedad_compuesta INT(11)
        REFERENCES tabla_compositor (campo_id)
        ON DELETE SET NULL
) ENGINE=InnoDB;
```

En el caso de propiedades compuestas en objetos relacionales, la eliminación se hará en cascada:

```
CREATE TABLE compositorcompuesto (
    compositorcompuesto_id INT(11) NOT NULL
```

```
      AUTO_INCREMENT PRIMARY KEY
  , compuesto INT(11) NOT NULL
  , FOREIGN KEY (compuesto)
        REFERENCES compuesto (compuesto_id)
        ON DELETE CASCADE
  , compositor INT(11) NOT NULL
  , FOREIGN KEY (compositor)
        REFERENCES compositor (compositor_id)
        ON DELETE CASCADE
  , relacion INT(3)
) ENGINE=InnoDB;
```

Todos los campos marcados como claves foráneas, serán idexados

Esto es, a fin de optimizar el rendimiento de la base de datos.

```
CREATE TABLE compositorcompuesto (
    compositorcompuesto_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY KEY
  , compuesto INT(11) NOT NULL
  , INDEX (compuesto)
  , FOREIGN KEY (compuesto)
        REFERENCES compuesto (compuesto_id)
        ON DELETE CASCADE
  , compositor INT(11) NOT NULL
  , INDEX (compositor)
  , FOREIGN KEY (compositor)
        REFERENCES compositor (compositor_id)
        ON DELETE CASCADE
  , relacion INT(3)
) ENGINE=InnoDB;
```

Esta indexación, deberá estar presente en cualquier campo indicado como clave foránea, de cualquier tabla.

Ten en cuenta que seguir todas las reglas de estilo antes mencionadas, será la forma más acertada de mantener un diseño simple.

Creación de la base de datos

Todas las consultas anteriores (obtenidas del mapeo relacional), deberán ser escritas en un archivo SQL (database.sql).

Al principio de este archivo, escribiremos también, la consulta SQL para crear y usar la base de datos:

```
CREATE DATABASE nuevadb;
USE nuevadb;
```

Es muy frecuente cometer errores de escritura que pueden afectar la sintaxis del lenguaje SQL, provocando que al ejecutar el archivo, la construcción de nuestra base de datos quede inconclusa.

Para prevenir este problema, como primera línea de nuestro *script* SQL, nos ocuparemos de eliminar previamente la base de datos. Esto nos permitirá ejecutar el archivo tantas veces como sea necesario, con solo corregir el error de sintaxis informado por MySQL.

```
DROP DATABASE IF EXISTS nuevadb;
CREATE DATABASE nuevadb;
```

```
USE nuevadb;
```

Finalmente, para correr el archivo y crear la base de datos, ejecutaremos el siguiente comando:

```
mysql -u root -p < database.sql
```

Si todo ha ido bien, ningún mensaje será mostrado en pantalla.

Para comprobar la base de datos, ingresar a MySQL:

```
mysql -u root -p
```

Mostrar todas las bases de datos para verificar que la nuestra haya sido creada:

```
show databases;
```

Si nuestra base de datos se encuentra creada, la seleccionamos para ver sus tablas:

```
use nuevadb;
show tables;
```

Para ver la estructura interna de cualquiera de las tablas, escribimos:

```
describe nombre_de_la_tabla;
```

Para salir, escribe:

```
exit
```

Capítulo VIII: Objetos, subtipos y herencia real

Como herencia entendemos a aquellos objetos que heredan de otros, adquiriendo así, sus mismas características. Pero a veces, se hace difícil discernir si realmente estamos en presencia de dos objetos diferentes, dos objetos similares o un solo objeto con una propiedad cuyo valor, marque la diferencia.

¿Herencia o propiedades distintivas?

Por ejemplo, si hablamos de azulejos y lajas, en principio parecería que estamos hablando de dos objetos diferentes. Sin embargo ambos objetos tienen las mismas propiedades: color, textura, espesor, anchura y longitud. Ambos objetos, a la vez, cumplen la misma función: servir de revestimiento a las paredes.

Pero color, textura, espesor, longitud y anchura ¿no son a caso las propiedades de un tipo de revestimiento? Este es el caso de un solo objeto

(Revestimiento) con una propiedad cuyo valor, marcará la diferencia: tipo, la cuál podrá tener diferentes valores (azulejo, laja y también cerámico, baldosa, etc.) que serán los que en definitiva, marquen la diferencia. Hablamos de un solo objeto con propiedades simples.

¿Herencia o polimorfismo?

Sin embargo, puedo tener dos objetos con las mismas propiedades y no tener la posibilidad de unificarlos, ya que los mismos, pertenecen a grupos de objetos diferentes: un vidrio también puede definirse con color, textura, espesor, anchura y longitud pero sin embargo, no pertenece al grupo de los revestimientos. Este es el caso de dos objetos diferentes, sin relación entre sí.

Herencia real y subtipos

En el afán de unificar objetos, podemos cometer errores que muchas veces pasan desapercibidos. Pensemos en el objeto pared: si una habitación tiene 4 paredes, una de las paredes tiene una ventana y otra de las paredes tiene una puerta

¿realmente estamos en presencia solo de un objeto pared?

Lo cierto e irrefutable es que “las cuatro son paredes”. De eso no hay ninguna duda. Pero la presencia de propiedades compuestas (ventana en un caso, puerta en el otro) nos está marcando claramente que estamos en presencia de subtipos del objeto.

Pared, tendrá como propiedades todas aquellas que hemos definido con anterioridad excepto puerta y ventana que pasarán a ser propiedades de los subtipos de pared: ParedConPuerta y ParedConVentana respectivamente.

Los subtipos heredarán todas las propiedades del tipo principal, incluso la propiedad objeto_id.

Por la base de datos, no habrá que preocuparse en hacer un nuevo mapeo, ya que cuando existe herencia, las propiedades de los subtipos comparten la tabla con la “clase madre”.

Definiendo la herencia en el código

Python:

```
class ClaseHija(ClaseMadre):
```

```
PHP:
class ClaseHija extends ClaseMadre { }
```

Inicializando subtipos

En los métodos constructores, los subtipos deben inicializarse no solo con sus nuevas propiedades, sino además, deberán inicializar (en su propio constructor), a la clase madre:

```
Python:
class ClaseHija(ClaseMadre):

    def __init__(self):
        super(ClaseHija, self).__init__()

PHP:
class ClaseHija extends ClaseMadre {

    function __construct() {
        parent::__construct();
    }
}
```

Finalmente, nuestras clases Pared, ParedConVentana y ParedConPuerta, se verán así:

```
Python:
class Pared(object):

    def __init__(self) :
        self.pared_id
        self.lajas = []
```

```
        self.habitacion = 0


class ParedConVentana(Pared):

    def __init__(self, ventana=None) :
        super(ParedConVentana, self).__init__()
        self.ventana = compose(ventana, Ventana)


class ParedConPuerta(Pared):

    def __init__(self, puerta=None) :
        super(ParedConPuerta, self).__init__()
        self.puerta = compose(puerta, Puerta)
```

PHP:

```
class Pared {

    function __construct() {
        $this->pared_id = 0;
        $this->lajas = array();
        $this->habitacion = 0;
    }
}


class ParedConVentana extends Pared {

    function __construct(Ventana $ventana) {
        parent::__construct();
        $this->ventana = $ventana;
    }
}


class ParedConPuerta extends Pared {

    function __construct(Puerta $puerta) {
        parent::__construct();
        $this->puerta = $puerta;
```

```
    }
}
```

Herencia: saber donde "parar"

Podríamos ponernos aún más meticulosos y decir que Puerta y Ventana son dos tipos de abertura diferentes y obtener entonces, el objeto ParedConAbertura en vez de ParedConVentana y ParedConPuerta. Sin embargo, aunque las aberturas compartan las mismas propiedades (como marco, altura, etc.), podríamos caer en una herencia infinita de abertura y el diseño de los objetos, estaría *"dejando de lado la simplicidad en nombre de la meticulosidad obsesiva"*.

Capítulo IX: Modelado de objetos y agrupación

En breves palabras (ya que no necesitamos decir mucho) y antes de comenzar a crear los métodos de nuestros objetos, una vez éstos han sido diseñados, estamos en condiciones de "modelarlos" y agruparlos a fin de mantener una estructura equilibrada y un diseño simple de la aplicación.

Modelado y agrupación

Se puede decir que un modelo es la forma de agrupar clases y categorizarlas.

Un modelo constituye un archivo (lo que para Python se denomina «módulo[12]»).

Cada modelo estará integrado por clases con relación directa:

- Cada clase estará en su propio modelo (el

12 No confundir el término con "módulo" en la Arquitectura de Software

cual, llevará el nombre de la clase "principal" en minúsculas)

- Subtipos estarán en el mismo modelo que el objeto madre

- Relacionales simples, estarán en el mismo modelo que el objeto al que componen

A la vez, todos los modelos, deberán agruparse en un directorio al que podremos llamar "models". En el caso de Python, este directorio deberá poder ser manipulado como un paquete. Por este motivo, un archivo __init__.py deberá incluirse al mismo.

Capítulo X: Bases de datos, consultas y abstracción

Para poder comenzar a trabajar con estructuras de datos, necesitaremos tener en cuenta que:

- Una clase, jamás debe conectarse de forma directa a la base de datos, puesto que resta simplicidad al diseño, al tiempo que provoca una gran redundancia de código, haciendo el sistema poco legible y difícil de mantener;

- Cada clase deberá recurrir entonces, a una capa de abstracción, que le permita conectarse a la base de datos y efectuar consultas;

Notar que por cuestiones de seguridad, en PHP trabajaremos con el conector mysqli y no con mysql. Sin embargo, en Python solo bastará con una función.

La capa de abstracción a bases de datos, pasará a formar parte del núcleo de la aplicación, puesto

que podrá ser reutilizada por cualquier componente de nuestro sistema.

Para albergar archivos de núcleo, incluiremos un directorio llamado “core”.

Creando una capa de abstracción en Python

En Python, no habrá demasiada complicación. Una simple función a la cual podamos pasarle nuestra consulta SQL, será suficiente.

Antes de comenzar: deberás instalar el módulo MySQLdb para poder conectarte a bases de datos MySQL:

```
sudo apt-get install python-mysqldb
```

Código de la capa de abstracción

Archivo: core/dblayer.py

```
# -*- coding: utf-8 -*-
import MySQLdb

from settings import DB_HOST, DB_USER, DB_PASS, \
      DB_NAME
```

```
def run_query(query):
    datos = [DB_HOST, DB_USER, DB_PASS, DB_NAME]
    conn = MySQLdb.connect(*datos)
    cursor = conn.cursor()
    cursor.execute(query)

    if query.upper().startswith('SELECT'):
        data = cursor.fetchall()
    else:
        conn.commit()
        if query.upper().startswith('INSERT'):
            data = cursor.lastrowid
        else:
            data = None

    cursor.close()
    conn.close()

    return data
```

Como puede observarse, la función run_query() recibe como parámetro una sentencia SQL. La función comprueba qué tipo de cláusula se está invocando:

- Cuando la sentencia sea de tipo SELECT, retornará una tupla con *N* tuplas dentro, equivalentes a la cantidad de registros obtenidos de la ejecución de dicha consulta
- Cuando la sentencia sea de tipo escritura (INSERT, DELETE y UPDATE), hará un *commit*.
- Y cuando se tratase de una sentencia de tipo INSERT, retornará la ID del último registro

insertado.

Creando una capa de abstracción en PHP con mysqli

En PHP, crear una capa de abstracción con el conector mysqli, es un asunto mucho más complejo que en Python y para nada equiparable.

El objetivo de utilizar el conector mysqli y no el tradicional conector mysql de PHP, es prevenir inyecciones SQL de forma simple y poder trabajar con aplicaciones mucho más robustas.

Pero ¿qué es exactamente mysqli y en qué se diferencia de mysql? Básicamente, como bien se define en el manual oficial de PHP, mysqli es "una extensión mejorada del conector mysql". Entre las principales diferencias, se encuentran además, sus dos grandes ventajas:

- Permite trabajar en estilo orientado a objetos (también continúa proveyendo soporte para estilo procedimental);
- Nos facilita una forma segura de filtrado de datos en sentencias SQL, para prevenir inyecciones SQL;

Sin dudas, mysqli es una gran ventaja frente al antiguo conector. Tiene una gran cantidad de clases, métodos, constantes y propiedades muy bien documentados[13]. Sin embargo, entender la documentación puede ser una tediosa tarea, en la cual, hallar un principio y un fin, se podrá convertir en la peor pesadilla de tu vida.

Antes de comenzar: deberás instalar el paquete php5-mysql para poder trabajar con el conector mysqli:

```
sudo apt-get install php5-mysql
```

Ahora sí. ¡Manos a la obra! Y a crear una capa de abstracción con mysqli orientado a objetos.

Recetario

Lo primero que debemos tener en cuenta, es que nuestra capa de abstracción deberá proveer de métodos públicos, que puedan ser llamados de forma estática, para que crear un objeto conector, no sea necesario.

---

13 http://php.net/manual/es/book.mysqli.php

Para poder lograr una capa de abstracción genérica, la clave es utilizar ReflectionClass[14] para crear una instancia de mysqli_stmt y mediante ReflectionClass->getMethod, invocar al método bind_param. De lo contrario, preparar una consulta SQL y enlazarle los valores a ser filtrados, será imposible.

Ten en cuenta que para seguir los ejemplos de este libro, es necesario contar con la versión 5.3.6 o superior, de PHP.

Propiedades

Nuestra capa de abstracción, tendrá una única propiedad pública, destinada a almacenar los datos obtenidos tras una consulta de selección. El resto de las propiedades, serán de ámbito protegido, accesibles solo desde nuestra clase y clases que hereden de ésta.

```
class DBOject {

    protected static $conn; # Objeto conector mysqli
    protected static $stmt; # preparación de la consulta SQL
    # Objeto Reflexivo de mysqli_stmt
    protected static $reflection;
    protected static $sql;  # Sentencia SQL a ser preparada
    # Array conteniendo los tipo de datos
    # más los datos a ser enlazados
```

14 http://php.net/manual/es/class.reflectionclass.php

```
    # (será  recibido como parámetro)
    protected static $data;
    # Colección de datos retornados por una consulta
    # de  selección
    public static $results;

}
```

La consulta SQL, deberá ser seteada en los modelos (clases) donde se requiera, incluyendo marcadores de parámetros (embebidos con el signo ?), en la posición donde un dato deba ser enlazado. Un ejemplo de ella, podría ser el siguiente:

```
$sql = "INSERT INTO vidrio
        (espesor, textura, pigmentacion)
        VALUES (?, ?, ?)";
```

Mientras que el array $data, deberá contener, como primer elemento, una string con el tipo de datos y los elementos siguientes, serán los datos a ser enlazados (todos pasados como *string*):

```
$data = array("iss",
              "{$this->espesor}",
              "{$this->textura}",
              "{$this->pigmentacion}");
```

El primer elemento, siempre representará el tipo de datos correspondiente al marcador de parámetro que se desea enlazar. Siendo los tipos de datos posibles: **s** (string), **i** (entero), **d** (doble) y **b** (blob).

Métodos

Conectar a la base de datos:

```
protected static function conectar() {
    self::$conn = new mysqli(DB_HOST, DB_USER,
                             DB_PASS, DB_NAME);
}
```

Requerirá 4 constantes predefinidas (se recomienda definirlas en un archivo settings): DB_HOST, DB_USER, DB_PASS y DB_NAME.

Preparar una sentencia SQL (con marcadores de parámetros):

```
protected static function preparar() {
    self::$stmt = self::$conn->prepare(
                                      self::$sql);
    self::$reflection = new ReflectionClass(
                                   'mysqli_stmt');
}
```

La clase ReflectionClass de PHP, cumple un papel fundamental: solo a través de ella podemos crear un objeto mysqli_stmt reflexivo, siendo ésta, la única alternativa que tenemos para preparar sentencias SQL con marcadores de parámetros, de forma dinámica.

La propiedad estática $sql (self::$sql) será creada

por el único método público que tendrá la clase.

Enlazar los datos con la sentencia SQL preparada:

```
protected static function set_params() {
    $method = self::$reflection->getMethod(
                                'bind_param');
    $method->invokeArgs(self::$stmt,
                        self::$data);
}
```

La propiedad estática $data que se pasa como segundo parámetro a invokeArgs, también será seteada por el único método público.

En este método (set_params), la variable temporal $method, llama reflexivamente (a través del método getMethod de reflectionClass), al método bind_param de la clase mysqli. En la siguiente instrucción, a través del método invokeArgs de ReflectionClass, le pasa por referencia a bind param, los datos a ser enlazados con la sentencia preparada (almacenados en el array $data). Podría decirse que invokeArgs, se comporta de forma similar a call_user_func_array().

Enlazar resultados de una consulta de selección:

```
protected static function get_data($fields) {
    $method = self::$reflection->getMethod(
```

```
                                    'bind_result');
        $method->invokeArgs(self::$stmt, $fields);
        while(self::$stmt->fetch()) {
            self::$results[] = unserialize(
                                 serialize($fields));
        }
    }
```

Este método es uno de los más “complejos y rebuscados”, que incluso cuenta con algunas “pseudo-magias” un tanto “raras” como el uso de las funciones serialize y unserialize en la la misma instrucción. Pero analicémoslo detenidamente.

El parámetro $fields será recibido a través del único método público que crearemos en nuestra capa de abstracción (este método, a la vez, recibirá este dato, también como parámetro, pero opcional).

Este parámetro, será un array asociativo, donde las claves, serán asociadas al nombre de los campos, y el valor de esas claves, al dato contenido en el campo.

Si tuviese la siguiente consulta SQL:

```
SELECT vidrio_id, espesor, textura, pigmentacion
FROM   vidrio
WHERE  vidrio_id = ?
```

Mi array asociativo, podría paracerse al siguiente:

```
$fields = array("vidrio_id" => "",
```

```
                "espesor" => "",
                "textura" => "",
                "pigmentacion" => "");
```

mysqli->bind_result() enlazará el campo vidro.espesor a la clave espesor, vidro.textura a la clave textura y vidrio.pigmentacion a la clave pigmentacion.

Las instrucciones:

```
$method = self::$reflection->getMethod(
                              'bind_result');
$method->invokeArgs(self::$stmt, $fields);
```

se comportan de forma similar, a sus homónimas en el método set_params. Llama reflexivamente al método bind_result de la clase mysqli y le pasa por referencia, al array $fields.

En el bucle while, estamos asociando iterativamente los datos obtenidos a nuestra propiedad pública $results. Pero ¿cómo lo hacemos? ¿para qué serializar y deserializar los datos?:

```
while(self::$stmt->fetch()) {
    self::$results[] = unserialize(
                           serialize($fields));
```

```
}
```

En cada iteración, stmt->fetch nos está retornando nuestro array $fields, asociado al registro de la tabla, sobre el cuál se está iterando. Es decir, que en cada iteración, stmt->fetch nos retornará algo como esto:

```
$fields = array("vidrio_id" => 1,
                "espesor" => 10,
                "textura" => "lisa",
                "pigmentacion" => "blancuzca");
```

Pero nuestro array $fields, ha sido pasado por referencia. Ergo, en cada iteración, su contenido será modificado.

Si a mi propiedad estática $results, le agrego como elemento, un array que está siendo modificado por referencia en el momento que lo estoy asignando a mi propiedad estática, mi propiedad estática, será también, modificada en cada iteración.

Para prevenir eso, *“freezo”* mi array $fields y lo almaceno en $results serializado. Pero como luego necesitaré recuperarlo, ahorro un paso y lo deserializo en la misma iteración. Al serializarlo, estoy “mágicamente” emulando una “inmutabilidad” de los datos asociados.

Notar que cuando la consulta efectuada se realice por coincidencia con una clave primaria (id del objeto), no será necesario acceder a la propiedad estática $results, ya que los resultados se encontrarán reflejados en el array $fields, quien al ser pasado por referencia, es modificado tras la llamada a esta capa de abstracción.

Cerrar conexiones abiertas:

```
protected static function finalizar() {
    self::$stmt->close();
    self::$conn->close();
}
```

Un método público que ejecute todas las acciones:

```
public static function ejecutar($sql, $data,
                                  $fields=False) {
    self::$sql = $sql;  # setear la propiedad $sql
    self::$data = $data;  # setear la propiedad $data
    self::conectar();  # conectar a la base de datos
    self::preparar();  # preparar la consulta SQL
    self::set_params();  # enlazar los datos
    self::$stmt->execute();  # ejecutar la consulta
    if($fields) {
        self::get_data($fields);
        return self::$results;
    } else {
        if(strpos(self::$sql,
                        strtoupper('INSERT')) === 0) {
            return self::$stmt->insert_id;
        }
    }
    self::finalizar();  # cerrar conexiones abiertas
```

```
}
```

La estructura de control de flujo condicional, que utiliza el método ejecutar(), es la encargada de discernir si se trata de una consulta de "lectura" a la base de datos para así, llamar al método get_data, o si se trata de una consulta de "escritura" (INSERT, UPDATE o DELETE).

En ese caso, verifica si es una escritura de tipo "INSERT" para retornar la id del nuevo registro creado.

Código completo de la capa de abstracción

```
class DBObject {

    protected static $conn;
    protected static $stmt;
    protected static $reflection;
    protected static $sql;
    protected static $data;
    public static $results;

    protected static function conectar() {
        self::$conn = new mysqli(DB_HOST, DB_USER, DB_PASS, DB_NAME);
    }

    protected static function preparar() {
        self::$stmt = self::$conn->prepare(self::$sql);
        self::$reflection = new ReflectionClass('mysqli_stmt');
    }

    protected static function set_params() {
        $method = self::$reflection->getMethod('bind_param');
        $method->invokeArgs(self::$stmt, self::$data);
    }

    protected static function get_data($fields) {
        $method = self::$reflection->getMethod('bind_result');
```

```
        $method->invokeArgs(self::$stmt, $fields);
        while(self::$stmt->fetch()) {
            self::$results[] = unserialize(serialize($fields));
        }
    }

    protected static function finalizar() {
        self::$stmt->close();
        self::$conn->close();
    }

    public static function ejecutar($sql, $data, $fields=False) {
        self::$sql = $sql;
        self::$data = $data;
        self::conectar();
        self::preparar();
        self::set_params();
        self::$stmt->execute();
        if($fields) {
            self::get_data($fields);
            return self::$results;
        } else {
            if(strpos(self::$sql, strtoupper('INSERT')) === 0) {
                return self::$stmt->insert_id;
            }
        }
        self::finalizar();
    }
}
```

¿Cómo utilizar la capa de abstracción creada?

En todos los casos, siempre será necesario invocar estáticamente al método `ejecutar()`, pasándole al menos dos parámetros: la sentencia SQL a preparar y un array con los datos a enlazar a la sentencia SQL preparada:

```
$sql = "INSERT INTO vidrio
        (espesor, textura, pigmentacion)
        VALUES (?, ?, ?)";

$data = array("iss",
              "{$this->espesor}",
```

```
              "{$this->textura}",
              "{$this->pigmentacion}");

$this->vidrio_id = DBObject::ejecutar($sql, $data);
```

Cuando se tratare de una consulta de selección, se deberá adicionar un tercer parámetro, el cuál será un array asociativo, cuyas claves, serán asociadas a los campos de la tabla:

```
$sql = "SELECT vidrio_id, espesor,
               textura, pigmentacion
        FROM   vidrio
        WHERE  vidrio_id = ?";

$data = array("i", "{$this->vidrio_id}");

$fields = array("vidrio_id" => "",
                "espesor" => "",
                "textura" => "",
                "pigmentacion" => "");

DBObject::ejecutar($sql, $data, $fields);
```

Capítulo XI: El método save() en objetos simples, compuestos y relacionales

La finalidad del método save() será guardar los datos de un objeto (nuevo o existente). Esto nos permitirá, poder recuperar el objeto más adelante, con su método get() correspondiente.

El método save() es uno de los más simples de todos los métodos (aunque sin dudas, el más simple de todos es el método destroy() que veremos más adelante).

Al ser su responsabilidad guardar un objeto, sea éste un nuevo objeto u objeto existente, tendrá que "ingeniárselas" para saber si debe ejecutar una consulta de inserción o de actualización.

Para ello, recurrirá a evaluar el valor de la propiedad objeto_id. Cuando ésta sea igual a cero, entonces un nuevo objeto ha sido creado. De lo contrario, un objeto existente, está solicitando guardar sus cambios.

Finalmente, la responsabilidad del método save(), será establecer el valor de la propiedad objeto_id con el dato retornado por la capa de abstracción, cuando la consulta, haya sido de tipo INSERT.

Se debe tener en cuenta que todos los métodos save() -sin excepción alguna-, para crear las consultas de inserción y actualización, tendrán en cuenta las mismas pautas que han sido mencionadas para el mapeo: todos los métodos save() guardarán solo y únicamente los datos de aquellas propiedades que han sido establecidas como campo de su propia tabla.

Si observamos los siguientes ejemplos, veremos que no existe diferencia alguna, entre los métodos save() de objetos simples, compuestos y relacionales:

Python:

```
# UN OBJETO SIMPLE
class Vidrio(object):

    def __init__(self):
        self.vidrio_id = 0
        self.espesor = 0
        self.textura = ''
        self.pigmentacion = ''

    def save(self):
        if self.vidrio_id == 0:
            sql = """INSERT INTO vidrio
                     (espesor, textura, pigmentacion)
```

```
                  VALUES (%i, '%s', '%s')""" % (
                     self.espesor, self.textura,
                     self.pigmentacion)

            self.vidrio_id = run_query(sql)
        else:
            sql = """UPDATE vidrio
                     SET espesor = %i,
                         textura = '%s',
                         pigmentacion = '%s'
                     WHERE vidrio_id = %i""" % (
                        self.espesor, self.textura,
                        self.pigmentacion,
                        self.vidrio_id)
            run_query(sql)


# UN OBJETO COMPUESTO
class Ventana(object):

    def __init__(self, marco):
        self.ventana_id = 0
        self.marco = compose(marco, Marco)

    def save(self):
        if self.ventana_id == 0:
            sql = """INSERT INTO ventana (marco)
                     VALUES (%i)""" % (
                        self.marco.marco_id)

            self.marco_id = run_query(sql)
        else:
            sql = """UPDATE ventana
                     SET marco = %i
                     WHERE ventana_id = %i""" % (
                        self.marco.marco_id,
                        self.ventana_id)
            run_query(sql)


# UN OBJETO RELACIONAL
class LajaPared(object):

    def __init__(self, compuesto, compositor=None):
```

```
        self.lajapared_id = 0
        self.pared = compose(compuesto, Pared)
        self.laja = compose(compositor, Laja)
        self.relacion = 1

    def set_relacion(self):
        tmpvar = 0
        while tmpvar < self.relacion:
            self.pared.add_laja(self.laja)
            tmpvar += 1

    def save(self):
        if self.lajapared_id == 0:
            sql = """INSERT INTO lajapared
                     (pared, laja, relacion)
                     VALUES (%i, %i, %i)""" % (
                        self.pared.pared_id,
                        self.laja.laja_id, self.relacion)
            self.lajapared_id = run_query(sql)
        else:
            sql = """UPDATE lajapared
                     SET pared = %i,
                         laja = %i,
                         relacion = %i
                     WHRE lajapared_id = %i""" % (
                        self.pared.pared_id,
                        self.laja.laja_id, self.relacion,
                        self.lajapared_id)
            run_query(sql)
```

PHP:

```
# UN OBJETO SIMPLE
class Vidrio{

    function __construct() {
        $this->vidrio_id = 0;
        $this->espesor = 0;
        $this->textura = '';
        $this->pigmentacion = '';
    }

    function save() {
```

```
        if($this->vidrio_id == 0) {
            $sql = "INSERT INTO vidrio
                    (espesor, textura, pigmentacion)
                    VALUES (?, ?, ?)";
            $data = array("iss",
                          "{$this->espesor}",
                          "{$this->textura}",
                          "{$this->pigmentacion}");
            $this->vidrio_id = DBObject::ejecutar(
                                         $sql, $data);
        } else {
            $sql = "UPDATE vidrio
                    SET espesor = ?,
                        textura = ?,
                        pigmentacion = ?
                    WHERE vidrio_id = ?";
            $data = array("issi",
                          "{$this->espesor}",
                          "{$this->textura}",
                          "{$this->pigmentacion}",
                          "{$this->vidrio_id}");
            DBObject::ejecutar($sql, $data);
        }
    }
}


# UN OBJETO COMPUESTO
class Ventana {

    function __construct(Marco $marco=NULL) {
        $this->ventana_id = 0;
        $this->marco = $marco;
    }

    function save() {
        if($this->ventana_id == 0) {
            $sql = "INSERT INTO ventana (marco)
                    VALUES (?)";
            $data = array("i",
                          "{$this->marco->marco_id}");
            $this->ventana_id = DBObject::ejecutar(
                                            $sql, $data);
        } else {
```

```
            $sql = "UPDATE ventana SET marco = ?
                    WHERE ventana_id = ?";
            $data = array("ii",
                          "{$this->marco->marco_id}",
                          "{$this->ventana_id}");
            DBObject::ejecutar($sql, $data);
        }
    }
}


# UN OBJETO RELACIONAL
class LajaPared {

    function __construct(Pared $compuesto,
                         Laja $compositor=NULL) {
        $this->lajapared_id = 0;
        $this->pared = $compuesto;
        $this->laja = $compositor;
        $this->relacion = 1;
    }

    function set_relacion() {
        $tmpvar = 0;
        while($tmpvar < $this->relacion) {
            $this->pared->add_laja($this->laja);
            $tmpvar++;
        }
    }

    function save() {
        if($this->lajapared_id == 0) {
            $sql = "INSERT INTO lajapared
                    (pared, laja, relacion)
                    VALUES (?, ?, ?)";
            $data = array("iii",
                          "{$this->pared->pared_id}",
                          "{$this->laja->laja_id}",
                          "{$this->relacion}");
            $this->lajapared_id = DBObject::ejecutar(
                                            $sql, $data);
        } else {
            $sql = "UPDATE lajapared
                    SET pared = ?,
```

```
                        laja = ?,
                        relacion = ?
                    WHERE lajapared_id = ?";
            $data = array("iiii",
                          "{$this->pared->pared_id}",
                          "{$this->laja->laja_id}",
                          "{$this->relacion}",
                          "{$this->lajapared_id}");
            DBObject::ejecutar($sql, $data);
        }
    }
}
```

Como se puede apreciar, no hay diferencias entre los métodos save() de los distintos tipos de objetos: todos los métodos save() siguen la misma lógica algorítmica.

El orden de llamada para los métodos save() siempre estará basado en el orden de dependencias.

El orden de llamada a los métodos save() es un factor clave en la creación de objetos, ya que ***de estos métodos depende*** *de forma directa,* ***la identidad*** *de cada uno* ***de nuestros objetos***

Llamar al método save() de un objeto sin

dependencias

Un objeto simple, basta con crearlo y llamar a su método save() correspondiente:

```
Python:
piso = Piso()
piso.tipo = 'flotante'
piso.material = 'madera'
piso.color = 'cedro'
piso.save()

PHP:
$piso = new Piso();
$piso->tipo = 'flotante';
$piso->material = 'madera';
$piso->color = 'cedro';
$piso->save();
```

Llamar al método save() de un objeto compuesto

Un objeto compuesto, deberá primero crear a su compositor, llamar al save() de éste, y luego crear el compuesto y llamar al save() del compuesto:

```
Python:
vidrio = Vidrio()
vidrio.espesor = 10
vidrio.textura = 'lisa'
vidrio.pigmentacion = 'blancuzca'
vidrio.save()

marco = Marco(vidrio)
marco.material = 'aluminio'
```

```
marco.color = 'blanco'
marco.save()
```

```
PHP:
$vidrio = new Vidrio();
$vidrio->espesor = 10;
$vidrio->textura = 'lisa';
$vidrio->pigmentacion = 'blancuzca';
$vidrio->save();

$marco = new Marco($vidrio);
$marco->material = 'aluminio';
$marco->color = 'blanco';
$marco->save();
```

Orden de llamada a los métodos save() cuando interviene un compositor de pertenencia

Cuando un objeto compuesto, dependa de un compositor de pertenencia, primero se deberá crear el compuesto y llamar a su save(). Luego, se crearán uno a uno los compositores de pertenencia llamando al save() del compositor en cada creación y una vez guardados todos los compositores, serán agregados al compuesto:

```
Python:
habitacion = Habitacion(piso, cielo_raso)
habitacion.save()

pared_izq = Pared()
pared_izq.habitacion = habitacion.habitacion_id
pared_izq.save()
```

```
habitacion.add_pared(pared_izq)
```

```
PHP:
$habitacion = new Habitacion($piso, $cielo_raso);
$habitacion->save();

$pared_izq = new Pared();
$pared_izq->habitacion = $habitacion->habitacion_id;
$pared_izq->save();

$habitacion->add_pared($pared_izq);
```

Orden de llamada de los métodos save() cuando interviene un compositor reutilizable con un objeto relacional

Cuando un objeto se componga de N compositores reutilizables, primero se deberá crear y guardar el compositor; luego se deberá crear y guardar el objeto compuesto(); a continuación, se creará el objeto relacional, se guardará la relación y finalmente, se se llamará al método set_relacion() del objeto relacional:

```
Python:
laja = Laja()
laja.color = 'beige'
laja.save()

pared = Pared()
pared.habitacion = habitacion.habitacion_id
pared.save()
```

```
lajapared = LajaPared(pared, laja)
lajapared.relacion = 75
lajapared.save()
lajapared.set_relacion()
```

PHP:

```
$laja = Laja();
$laja->color = 'beige';
$laja->save();

$pared = Pared();
$pared->habitacion = $habitacion->habitacion_id;
$pared->save();

$lajapared = LajaPared($pared, $laja);
$lajapared->relacion = 75;
$lajapared->save();
$lajapared->set_relacion();
```

Capítulo XII: El método destroy()

El método destroy() -destinado a destruir un objeto existente haciéndolo irrecuperable-, es el más simple de todos los métodos.

Se encuentra presente en todos los objetos, excepto en los relacionales multiplicadores, ya que estos métodos no necesitan destruir -ni deben hacerlo- una relación, sino que solo deberían poder modificarla. Puesto que la relación, solo puede destruirse -con efecto en cascada- cuando uno de los “integrantes de la relación” sea destruido.

Este método entonces, solo creará una sentencia SQL con la cláusula DELETE identificando los datos a destruir, por medio de la ID del objeto.

Python:

```
def destroy(self):
    sql = """DELETE FROM obj
             WHERE obj_id = %i""" % self.obj_id
    run_query(sql)
    self.obj_id = 0
```

PHP:

```
function destroy() {
    $sql = "DELETE FROM obj WHERE obj_id = ?";
    $data = array("i", "{$this->obj_id}");
    DBObject::ejecutar($sql, $data);
    $this->obj_id = 0;
}
```

Capítulo XIII: Métodos get() estándar, para objetos simples y objetos compuestos

El método get() es el método más complejo de los tres métodos comunes, save(), get() y destroy().

El método get() en objetos simples

En los objetos sin dependencias (es decir, objetos simples), el método get() solo realizará una consulta a la base de datos para obtener los valores que le ayuden a *setear* sus propiedades:

Python:

```
class Vidrio(object):

    def __init__(self):
        self.vidrio_id = 0
        self.espesor = 0
        self.textura = ''
        self.pigmentacion = ''
```

```
    def get(self):
        sql = """SELECT vidrio_id, espesor,
                        textura, pigmentacion
                 FROM vidrio
                 WHERE vidrio_id = %i""" % \
                                     self.vidrio_id

        fields = run_query(sql)[0]
        self.vidrio_id = fields[0]
        self.espesor = fields[1]
        self.textura = fields[2]
        self.pigmentacion = fields[3]
```

PHP:

```
class Vidrio{

    function __construct() {
        $this->vidrio_id = 0;
        $this->espesor = 0;
        $this->textura = '';
        $this->pigmentacion = '';
    }

    function get() {
        $sql = "SELECT vidrio_id, espesor,
                       textura, pigmentacion
                FROM vidrio
                WHERE vidrio_id = ?";
        $data = array("i", "{$this->vidrio_id}");
        $fields = array("vidrio_id" => "",
                        "espesor" => "",
                        "textura" => "",
                        "pigmentacion" => "");
        DBObject::ejecutar($sql, $data, $fields);
        $this->vidrio_id = $fields['vidrio_id'];
        $this->espesor = $fields['espesor'];
        $this->textura = $fields['textura'];
        $this->pigmentacion = $fields['pigmentacion'];
    }
```

El método get() en objetos compuestos

En los objetos con propiedades compuestas por un único objeto, el método get() estará dividido en dos partes: la primera parte actuará como el get() de un objeto simple, mientras que la segunda, recurrirá al get() de su compositor, para *setear* las propiedades compuestas por un solo objeto:

Python:

```
class Marco(object):

    def __init__(self, vidrio=None):
        self.marco_id = 0
        self.material = ''
        self.color = ''
        self.vidrio = compose(vidrio, Vidrio)

    def get(self):
        sql = """SELECT marco_id, material,
                        color, vidrio
                 FROM marco
                 WHERE marco_id = %i""" % self.marco_id
        fields = run_query(sql)[0]
        self.marco_id = fields[0]
        self.material = fields[1]
        self.color = fields[2]

        vidrio = Vidrio()
        vidrio.vidrio_id = fields[3]
        vidrio.get()

        self.vidrio = vidrio
```

PHP:

```
class Marco {
```

```
    function __construct(Vidrio $vidrio=NULL) {
        $this->marco_id = 0;
        $this->material = '';
        $this->color = '';
        $this->vidrio = $vidrio;
    }

    function get() {
        $sql = "SELECT marco_id, material,
                       color, vidrio
                FROM marco
                WHERE marco_id = ?";
        $data = array("i", "{$this->marco_id}");
        $fields = array("marco_id" => "",
                        "material" => "",
                        "color" => "",
                        "vidrio" => "");
        DBObject::ejecutar($sql, $data, $fields);
        $this->marco_id = $fields['marco_id'];
        $this->material = $fields['material'];
        $this->color = $fields['color'];

        $vidrio = new Vidrio();
        $vidrio->vidrio_id = $fields['vidrio'];
        $vidrio->get();

        $this->vidrio = $vidrio;
    }

}
```

Más adelante, veremos como con la ayuda de una fábrica de objetos, todo el código resaltado (es decir, la creación del compositor), pasará a ser responsabilidad de un nuevo objeto Factory.

Es válido aclarar que los subtipos y sus clases madres, poseerán -cada uno de ellos-, un método get() estándar como cualquier otro objeto.

Capítulo XIV: Los métodos get() en objetos compositores de pertenencia

Cuando un objeto sea considerado compositor de pertenencia (como por ejemplo, Pared), éste, contará con su método get() estándar más un método get auxiliar, cuya única responsabilidad, será la de proveer una colección de datos compuestos por la identidad de cada uno de los compositores que pertenecen al mismo compuesto.

Por favor, téngase en cuenta que este método get auxiliar, no retornará "objetos" sino por el contrario, solo retornará datos que sirvan al objeto compuesto para obtener sus compositores exclusivos.

Vale aclarar que **cuando los objetos de pertenencia sean subtipos de un objeto madre, el método get auxiliar, siempre deberá pertenecer a la clase principal.**

Los métodos get auxiliares, seleccionan solo la identidad de los compositores, utilizando como condición, el valor de la propiedad de pertenencia.

Cuando el compositor en cuestión, posea subtipos, además de la identidad del objeto, deberá retornar True o False en correspondencia a cada subtipo.

Un ejemplo de consulta SQL para el objeto madre Pared, podría ser el siguiente:

```
SELECT   pared_id,
         IF(ventana IS NULL, False, True),
         IF(puerta IS NULL, False, True)

FROM     pared

WHERE    habitacion = N
```

El nombre de dichos métodos estará conformado por la palabra get_ seguida del nombre de la propiedad del objeto compuesto, a la cual dicho compositor compone. Por ejemplo, el método auxiliar de la clase Pared se llamará get_paredes().

Un ejemplo de método get auxiliar, podríamos encontrarlo en la clase madre Pared:

Python:

```
def get_paredes(self):
    sql = """SELECT pared_id,
                    IF(ventana IS NULL, False, True),
                    IF(puerta IS NULL, False, True)
             FROM pared
```

```
                 WHERE habitacion = %i""" % self.habitacion

    return run_query(sql)
```

PHP:
```
function get_paredes() {
    $sql = "SELECT pared_id,
                   IF(ventana IS NULL, False, True),
                   IF(puerta IS NULL, False, True)
            FROM pared WHERE habitacion = ?";
    $data = array("i", "{$this->habitacion}");
    $fields = array("pared_id" => "");
    return DBObject::ejecutar(
                         $sql, $data, $fields);
}
```

Como se puede observar, los métodos get auxiliares solo retornan datos y no objetos. En el caso de Python, retornará una tupla con *N* tuplas dentro, conteniendo la identidad de cada compositor pertenenciente al compuesto que luego lo invoque. Y en el caso de PHP, retornará un array, compuesto de *N* arrays asociativos.

Estos métodos get auxiliares, como se comentó en el párrafo anterior, serán invocados desde el método get() del objeto compuesto, a quien le tocará la parte más compleja “en la historia de los métodos get() no estándar”. Veamos el caso de Habitacion:

Python:
```
def get(self):
```

```
sql = """SELECT habitacion_id,
                piso, cielo_raso
         FROM   habitacion
         WHERE
          habitacion_id = %i""" % self.habitacion_id
fields = run_query(sql)
self.habitacion_id = fields[0]

piso = Piso()
piso.piso_id = fields[1]
piso.get()
self.piso = piso

cr = CieloRaso()
cr.cielo_raso_id = fields[2]
cr.get()
self.cielo_raso = cr

# Hasta aquí el método get() estándar
# Incorporamos una extención

pared = Pared()
pared.habitacion = self.habitacion_id

# llamada al método auxiliar
paredes = pared.get_paredes()

# Iteramos para obtener los objetos
for tupla in paredes:
    if tupla[1] is not False:
        pared = ParedConVentana()
        pared.pared_id = tupla[0]
        pared.get()
        self.add_paredconventana(pared)
    elif tupla[2] is not False:
        pared = ParedConPuerta()
        pared.pared_id = tupla[0]
        pared.get()
        self.add_paredconpuerta(pared)
    else:
        pared = Pared()
        pared.pared_id = tupla[0]
```

```
            pared.get()
            self.add_pared(pared)
```

PHP:

```
function get() {
    $sql = "SELECT habitacion_id,
                   piso, cielo_raso
            FROM   habitacion
            WHERE  habitacion_id = ?"

    $data = array("i", "{$this->habitacion_id}");
    $fields = array("habitacion_id" => "",
                    "piso" => "",
                    "cielo_raso" => "",);

    DBObject::ejecutar($sql, $data, $fields);

    $this->habitacion_id = $fields['habitacion_id'];

    $piso = new Piso();
    $piso->piso_id = $fields['piso'];
    $piso->get();
    $this->piso = $piso;

    $cr = new CieloRaso();
    $cr->cielo_raso_id = $fields['cielo_raso'];
    $cr->get();
    $this->cielo_raso = $cr;

    # Hasta aquí el método get() estándar
    # Incorporamos una extensión

    $pared = new Pared();
    $pared->habitacion = $this->habitacion_id;

    # llamada al método auxiliar
    $paredes = $pared->get_paredes();

    # Iteramos para obtener los objetos
    foreach($paredes as $array) {
```

```
        if($array['ventana'] !== False) {
            $pared = new ParedConVentana();
            $pared->pared_id = $array['pared_id'];
            $pared->get();
            $this->add_paredconventana($pared);
        } elseif($array['puerta'] !== False) {
            $pared = new ParedConPuerta();
            $pared->pared_id = $array['pared_id'];
            $pared->get();
            $this->add_paredconpuerta($pared);
        } else {
            $pared = new Pared();
            $pared->pared_id = $array['pared_id'];
            $pared->get();
            $this->add_pared($pared);
        }
    }
}
```

Capítulo XV: El método get() de los objetos relacionales multiplicadores

Mientras que **los objetos compositores reutilizables contarán con un método get() estándar**, los objetos relacionales contarán con método get() apenas modificado.

En principio, debemos tener en cuenta que los métodos get() de los objetos relacionales, solo serán llamados desde el método get() de un objeto compuesto, quien a la vez, se deberá pasar a sí mismo como parámetro al momento de crear el objeto relacional.

El método get() de un relacional multiplicador, obtendrá los valores necesarios para configurar todas las propiedades, estableciendo como condición, la identidad del objeto al que compone.

Tras obtener los datos de la base de datos, actuará como cualquier otro método get() estándar:

1. Modificará su propiedad relación
2. Recuperará al compositor (invocando al método get() de éste) para modificar a su propiedad “compositor”

Finalmente, llamará a su propio método set_relacion() para que se encargue de componer al objeto que lo ha invocado.

Un ejemplo de consulta SQL de selección en un objeto relacional multiplicador, sería el siguiente:

```
SELECT relacional_id, compositor, relacion
FROM   relacional
WHERE  compuesto = N
```

De modo genérico, el método get() de un relacional multiplicador, deberá guardar siempre la siguiente forma y estructura:

Python:

```
def get(self):
    sql = """SELECT relacional_id,
                    compositor, relacion
             FROM   relacional
             WHERE  compuesto = %i""" % \
                self.compuesto.compuesto_id
    fields = run_query(sql)[0]
    self.relacion = fields[1]

    compositor = Compositor()
    compositor.compositor_id = fields[0]
    compositor.get()
```

```
    self.objetocompositor = compositor
    self.set_relacion()
```

PHP:

```
function get() {
    $sql = "SELECT relacional_id,
                   compositor, relacion
            FROM   relacional
            WHERE  compuesto = N";
    $data = array("i",
                  "{$this->compuesto->compuesto}");
    $fields = array("relacional_id" => "",
                    "compositor" => "",
                    "relacion" => "");
    DBObject::ejecutar($sql, $data, $fields);
    $this->relacion = $fields['relacion'];

    $compositor = new Compositor();
    $compositor->compositor_id = $fields['compositor'];
    $compositor->get();

    $this->compositor = $compositor;
    $this->set_relacion();
}
```

Finalmente, el objeto compuesto se encargará de llamar al get() del relacional, como última instancia de su método get() estándar:

Python:

```
def get(self):
    ... pasos estándar
    relacional = Relacional(self)
    relacional.get()
```

```
PHP:
function get():
    ... pasos estándar
    $relacional = new Relacional($this);
    $relacional->get();
```

Notar que al crear el objeto relacional, el objeto compuesto se pasa a sí mismo como parámetro:

```
Python: Relacional(self)
   PHP: new Relacional($this)
```

Capítulo XVI: Factoría de objetos con Factory Pattern. Objetos compuestos con métodos get() mucho más simples

Factory es -al igual que *composite*- un patrón de diseño. Su sencillez y simpleza, consisten en un método encargado de crear otros objetos.

¿Para qué nos puede ser útil un objeto Factory? Para que los métodos get() que deben llamar al método homónimo de los objetos que lo componen, pueda ser más limpio, menos redundante y por lo tanto, más eficiente.

Imaginemos un objeto compuesto por otros 3 objetos. El método get() de éste, se verá como el siguiente:

Python:

```
class ObjetoD(object):

    def __init__(self, objetoa=None, objetob=None,
```

```
                 objetoc=None):
        self.objetod_id = 0
        self.objetoa = compose(objetoa, ObjetoA)
        self.objetob = compose(objetob, ObjetoB)
        self.objetoc = compose(objetoc, ObjetoC)

    def get(self):
        sql = """SELECT objetod_id, objetoa,
                 objetob, objetoc
                 FROM   objetod
                 WHERE  objetod_id = %i""" % self.objetod_id

        fields = run_query(sql)[0]

        self.objetod_id = fields[0]

        objetoa = ObjetoA()
        objetoa.objetoa_id = fields[1]
        objetoa.get()
        self.objetoa = objetoa

        objetob = ObjetoB()
        objetob.objetob_id = fields[2]
        objetob.get()
        self.objetob = objetob

        objetoc = ObjetoC()
        objetoc.objetoc_id = fields[3]
        objetoc.get()
        self.objetoc = objetoc
```

PHP:

```
class ObjetoD {

    function __construct(ObjetoA $objetoa=NULL,
                         ObjetoB $objetob=NULL,
                         ObjetoC $objetoc=NULL) {
        $this->objetod_id = 0;
        $this->objetoa = $objetoa;
        $this->objetob = $objetob;
        $this->objetoc = $objetoc;
    }


    function get() {
```

```
        $sql = "SELECT objetod_id, objetoa,
                       objetob, objetoc
                FROM   objetod
                WHERE  objetod_id = ?";

        $data = array("i", "{$this->objetod_id}");

        $fields = array("objetod_id" => "",
                        "objetoa" => "",
                        "objetob" => "",
                        "objetoc" => "");

        DBObject::ejecutar($sql, $data, $fields);

        $this->objetod_id = $fields['objetod_id'];

        $objetoa = new ObjetoA();
        $objetoa->objetoa_id = $fields['objetoa'];
        $objetoa->get();
        $this->objetoa = $objetoa;

        $objetob = new ObjetoB();
        $objetob->objetob_id = $fields['objetob'];
        $objetob->get();
        $this->objetob = $objetob;

        $objetoc = new ObjetoC();
        $objetoc->objetoc_id = $fields['objetoc'];
        $objetoc->get();
        $this->objetoc = $objetoc;
    }

}
```

Hay una realidad y es que el método get() del ObjetoD solo debería tener la responsabilidad de *setear* sus propiedades con los datos obtenidos. Sin embargo, como puede verse, se encuentra forzado a tener que crear una instancia de cada una de los objetos que lo componen, modificar su id, llamar al

método get() de cada uno y finalmente, poder generar la composición necesaria.

Un objeto a nivel del *core*, encargado de crear y retornar otros objetos, será la solución a este problema:

Python:
```
class FactoryClass(object):

    def make(cls, clase, id_value, idname=''):
        obj = clase()
        p = idname if idname else "%s_id" % \
            obj.__class__.__name__.lower()
        setattr(obj, p, id_value)
        obj.get()
        return obj
```

PHP:
```
class FactoryClass {

    public static function make($cls, $id_value,
                                $idname='') {
        $p = ($idname) ? $idname : strtolower($cls) . "_id";
        $obj = new $cls();
        $obj->$p = $id_value;
        $obj->get();
        return $obj;
    }
}
```

De esta forma, el *seteo* de las propiedades compuestas de un objeto, requerirá sólo de una

llamada estática al método make() de FactoryClass:

Python:

```
class ObjetoD(object):

    def __init__(self, objetoa=None, objetob=None,
                 objetoc=None):
        self.objetod_id = 0
        self.objetoa = objetoa
        self.objetob = objetob
        self.objetoc = objetoc

    def get(self):
        sql = """SELECT objetod_id, objetoa,
                 objetob, objetoc
                 FROM   objetod
                WHERE  objetod_id = %i""" % self.objetod_id

        fields = run_query(sql)[0]
        self.objetod_id = fields[0]
        self.objetoa = FactoryClass().make(
                                     ObjetoA, fields[1])
        self.objetob = FactoryClass().make(
                                     ObjetoB, fields[2])
        self.objetoc = FactoryClass().make(
                                     ObjetoC, fields[3])
```

PHP:

```
class ObjetoD {

    function __construct(ObjetoA $objetoa=NULL,
                         ObjetoB $objetob=NULL,
                         ObjetoC $objetoc=NULL) {
        $this->objetod_id = 0;
        $this->objetoa = $objetoa;
        $this->objetob = $objetob;
        $this->objetoc = $objetoc;
    }


    function get() {
        $sql = "SELECT objetod_id, objetoa,
                       objetob, objetoc
                FROM   objetod
```

```
                WHERE  objetod_id = ?"

        $data = array("i", "{$this->objetod_id}");

        $fields = array("objetod_id" => "",
                        "objetoa" => "",
                        "objetob" => "",
                        "objetoc" => "");

        DBObject::ejecutar($sql, $data, $fields);
        $this->objetod_id = $fields['objetod_id'];
        $this->objetoa = FactoryClass::make(
                             'ObjetoA', $fields['objetoa']);
        $this->objetob = FactoryClass::make(
                             'ObjetoB', $fields['objetob']);
        $this->objetoc = FactoryClass::make(
                             'ObjetoC', $fields['objetoc']);
    }

}
```

Capítulo XVII: Objetos colectores de instancia única y Singleton Pattern

Uno de los errores más graves y a la vez más frecuente que todos los programadores solemos cometer, es diseñar nuestros objetos pensando en cómo éstos deberán mostrarse al usuario en una interfaz gráfica. De allí, que en una gran cantidad de oportunidades, nos encontraremos con un objeto que define un método destinado a retornar una colección de objetos de sí mismo.

Pero éste, ¡es un error garrafal! Un objeto es 1 solo objeto y no, una colección de objetos. Por lo tanto una colección de objetos es un «ObjetoAColeccion» compuesto de N objetos A.

«La colección de objetos A tiene varios objetos A»

Python:

```
class ObjetoACollection(object):
```

```
    def __init__(self):
        self.objetosa = []
```

PHP:

```
class ObjetoACollection {

    function __construct() {
        $this->objetosa = array();
    }
}
```

Sin embargo, solo puede existir una -y solo una- colección de objetos A. Esta colección será la suma de todos los objetos A existentes, y por tanto, no será necesaria una ID de ObjetoACollection. Pero debemos asegurarnos de que esto se cumpla. Y para ello, diseñaremos al nuevo ObjetoCollection como un Singleton.

Singleton es un patrón de diseño. *Un Singleton es un objeto de instancia única (no puede ser instanciado más de una vez) y para lograr esto, él -y solo él-, puede generar una instancia de sí mismo.*

Características de un Singleton colector en PHP

Sólo él puede crear una instancia de sí mismo. Por lo tanto, tendrá un método constructor privado que impida que el objeto pueda ser instanciado desde un ámbito diferente a la propia clase.

```
class ObjectCollection {

    private function __construct() {
        $this->objects = array();
    }
}
```

Como solo podrá tener una única instancia, la misma se almacenará en una propiedad privada estática:

```
class ObjectCollection {

    private static $objectcollection;
    # almacenará una instancia de sí mismo

    private function __construct() {
        $this->objects = array();
    }
}
```

Deberá contar con un método de agregación privado:

```
class ObjectCollection {

    private static $objectcollection;

    private function __construct() {
        $this->objects = array();
    }

    private function add_object(Object $object) {
        $this->objects[] = $object;
    }
}
```

El único método público, será su propio get() estático quien antes de actuar, será el encargado de crear una única instancia de sí mismo:

```
class ObjectCollection {

    private static $objectcollection;

    private function __construct() {
        $this->objects = array();
    }

    private function add_object(Object $object) {
        $this->objects[] = $object;
    }

    public static function get() {
        if(empty(self::$objectcollection)) {
            self::$objectcollection = new ObjectCollection();
        }
    }
}
```

Finalmente, deberá retornar una instancia de sí mismo:

```
class ObjectCollection {

    private static $objectcollection;

    private function __construct() {
        $this->objects = array();
    }

    private function add_object(Object $object) {
        $this->objects[] = $object;
    }

    public static function get() {
        if(empty(self::$objectcollection)) {
            self::$objectcollection = new ObjectCollection();
        }
    }

    return self::$objectcollection;

}
```

Características de un Singleton colector en Python

Sólo él puede crear una instancia de sí mismo. Por lo tanto, tendrá un método constructor privado que impida que el objeto pueda ser instanciado desde un ámbito diferente a la propia clase. En Python, no existe un método constructor privado. Por lo tanto, habrá que sobrescribir el método __new__ heradado de object. A la vez, dicha instancia,

deberá almacenarse en una propiedad privada:

```
class ObjectCollection(object):

    __objectcollection = None

    def __new__(cls):
        if cls.__objectcollection is None:
            cls.__objectcollection = super(
               ObjectCollection, cls).__new__(cls)
        return cls.__objectcollection
```

El método __new__ es un método estático invocado por Python al crear una nueva instancia de clase. Al sobrescribirlo, se debe invocar al método __new__ de la superclase, utilizando super(clase_actual, cls).__new__(cls). Cada vez que el objeto colector sea llamado, el mismo objeto será retornado (no se crearán múltiples instancias). Es decir que la sobre-escritura del método __new__, será la encargada de retornar siempre, la misma instancia.

El método __init__ será suprimido, a fin de forzar la creación del objeto solo y únicamente dentro de la propia clase. En reemplazo del método __init__ un método set() privado, será el encargado de definir las propiedades del colector:

```
class ObjectCollection(object):

    __objectcollection = None
```

```
    def __new__(cls):
        if cls.__objectcollection is None:
            cls.__objectcollection = super(
               ObjectCollection, cls).__new__(cls)
            return cls.__objectcollection

    def __set(self):
        self.__objectcollection.objetos = []
```

Deberá contar con un método de agregación privado:

```
class ObjectCollection(object):

    __objectcollection = None

    def __new__(cls):
        if cls.__objectcollection is None:
            cls.__objectcollection = super(
               ObjectCollection, cls).__new__(cls)
            return cls.__objectcollection

    def __set(self):
        self.__objectcollection.objetos = []

    def __add_objeto(self, objeto):
        self.__objectcollection.objetos.append(
            objeto)
```

El único método público, será su propio get() quien antes de actuar, será el encargado de llamar a __set():

```
class ObjectCollection(object):
```

```
    __objectcollection = None

    def __new__(cls):
        if cls.__objectcollection is None:
            cls.__objectcollection = super(
               ObjectCollection, cls).__new__(cls)
            return cls.__objectcollection

    def __set(self):
        self.__objectcollection.objetos = []

    def __add_objeto(self, objeto):
        self.__objectcollection.objetos.append(
            objeto)

    def get(self):
        self.__objectcollection.__set()
```

Finalmente, deberá retornar una instancia de sí mismo:

```
class ObjectCollection(object):

    __objectcollection = None

    def __new__(cls):
        if cls.__objectcollection is None:
            cls.__objectcollection = super(
               ObjectCollection, cls).__new__(cls)
            return cls.__objectcollection

    def __set(self):
        self.__objectcollection.objetos = []

    def __add_objeto(self, objeto):
        self.__objectcollection.objetos.append(
            objeto)
```

```
    def get(self):
        self.__objectcollection.__set()
        return self.__objectcollection
```

El método get() del singleton colector

El método get() de este singleton, realizará una consulta a la base de datos, trayendo todos los objetos de la colección que encuentre e iterando sobre esos resultados, almacenará cada objeto de la colección creado con FactoryClass:

Python:

```
def get(self):
    self.__objectcollection.__set()

    sql = "SELECT objeto_id FROM objeto"
    fields = run_query(sql)

    for field in fields:
        self.__objectcollection.__add_objeto(
            FactoryClass().make(
                Objeto, field[0]))

    return self.__objectcollection
```

PHP:

```
public static function get() {
    if(empty(self::$objectcollection)) {
        self::$ objectcollection = new ObjectCollection();
    }
```

```
    $sql = "SELECT object_id FROM object
            WHERE object_id > ?";
    $data = array("i", "0");
    $fields = array("object_id" => "");
    DBObject::ejecutar($sql, $data, $fields);

    foreach(DBObject::$results as $array) {
        self::$objectcollection->add_object(
                FactoryClass::make(
                           'Object',
                           $array['object_id']));
    }

    return self::$objectcollection;
}
```

Capítulo XVIII: Objetos relacionales complejos (conectores lógicos relacionales)

Mis alumnos conocen a estos objetos por un nombre bastante particular, el cuál hemos decidido adoptar debido a la "complejidad" de los mismos. Muy probablemente, cuando llegues al final del capítulo, tu solo podrás imaginar qué nombre le hemos otorgado con mis alumnos, a este tipo de objetos.

A lo largo del capítulo y, a fin de evitar confundirnos con los objetos relacionales simples vistos anteriormente, me referiré a este tipo de objetos como *«conectores lógicos relacionales»* o simplemente *«conectores lógicos»*.

Un conector lógico es una especie de mezcla entre compositor de pertenencia y objeto relacional simple.

Su finalidad, es la de establecer la conexión lógica existente entre N_1 objetos compositores

reutilizables de identidad diferenciada y N_2 objetos compuestos.

Recientemente, Christian -uno de mis alumnos-, me trajo un caso de compositores reutilizables que requieren de un conector lógico, ampliamente mucho más esclarecedor que el que venía utilizando como ejemplo en mis clases. Por ese motivo, me tomo la atribución de citarlo en este libro como ejemplo.

Imaginemos un sistema de gestión de eventos festivos, donde tenemos un amplio listado de invitados. Cada invitado puede asistir a diferentes eventos, lo que significa que los distintos eventos, pueden compartir los mismos invitados.

Al mismo tiempo, cada evento tiene un gran número de invitados.

Si los invitados pueden asistir a diferentes eventos, descartamos estar en presencia de compositores de pertenencia. Y al estar `Evento` compuesto por varios invitados, pero cada uno de ellos, de identidad única, descartamos estar en presencia de un compositor reutilizable cuya relación, requiera ser establecida por un objeto relacional multiplicador.

Tenemos en consecuencia una relación de *N* compositores a *N* compuestos. ¿Cómo haremos

para establecer la relación entre ellos?

Es aquí que surge la necesidad de contar con un conector lógico relacional.

Propiedades del conector lógico

El objeto conector lógico, contará entonces, con al menos tres propiedades:

1. Su propiedad objeto_id
2. Objeto compuesto
3. Una colección de compositores cuyo valor, se obtendrá de la colección provista por el objeto compuesto

Python:
```
class InvitadoEvento(object):

    def __init__(self, evento):
        self.invitadoevento_id = 0
        self.evento = compose(evento, Evento)
        self.invitados = evento.invitados
```

PHP:
```
class InvitadoEvento {

    function __construct(Evento $evento) {
        $this->invitadoevento_id = 0;
        $this->evento = $evento;
        $this->invitados = $evento->invitados;
    }
```

```
}
```

Mapeo relacional

Cuando un conector lógico debe ser mapeado, se tendrá en consideración todo lo visto en el Capítulo VII. Pero ¿cómo se mapean las colecciones?

La propiedad colectora de este objeto, a diferencia de las vistas hasta ahora, no posee como valor asociado un array ni una lista vacía. Por el contrario, desde su inicialización, adoptará como valor una colección de objetos.

El conector lógico deberá iterar sobre la propiedad colectora y por tal motivo, dichas propiedades son mapeadas como si se tratara del objeto compositor en un relacional simple. Obteniendo por tanto (si continuamos con el mismo ejemplo) un campo "invitado" que será clave foránea con efecto en cascada.

Por consiguiente, la consulta SQL generada, lucirá como la siguiente:

```
CREATE TABLE invitadoevento (
  invitadoevento_id INT(11) NOT NULL
      AUTO_INCREMENT PRIMARY_KEY
  , evento INT(11)
  , INDEX(evento)
```

```
    , FOREIGN KEY (evento)
        REFERENCES evento (evento_id)
        ON DELETE CASCADE
    , invitado INT(11)
    , INDEX(invitado)
    , FOREIGN KEY (invitado)
        REFERENCES invitado (invitado_id)
        ON DELETE CASCADE
) ENGINE= InnoDB;
```

Los métodos save(), get() y destroy() del conector lógico

El conector lógico contará con tres métodos, similares en concepto a los métodos estándar ya conocidos, pero diferentes en su algorítmica.

El método save()

El método save() del objeto conector es quien contará con el algoritmo más complejo de todos los métodos.

A diferencia de los métodos save() estándar, éstos solo deberán guardar nuevas relaciones, puesto que a nivel conceptual no existe la posibilidad de modificar una relación existente (en este tipo de objetos), sino que la misma debe ser destruida y

reemplazada por una nueva. Por tal motivo, siempre antes de ejecutar su propio algoritmo, llamarán al método de destrucción. Ten en cuenta que la relación final resultante, siempre será entre 1 compositor y 1 compuesto.

Dicho método deberá generar de forma dinámica, tanto la consulta SQL de inserción como los valores a ser insertados, ya que deberá guardar cada par compuesto-compositor como un único registro y no queremos conectarnos y desconectarnos a la base de datos, por cada par compuesto-compositor a insertar.

Para lograrlo, deberá iterar sobre la propiedad colectora:

Python:
```
def save(self):
    self.destroy()
    sql = """INSERT INTO invitadoevento
             (evento, invitado)"""

    data = []
    tmpvar = 0

    for invitado in self.invitados:
        sql += ", " if tmpvar > 0 else " VALUES "
        sql += "(%i, %i)"
        data.append(self.evento.evento_id)
        data.append(invitado.invitado_id)
        tmpvar += 1

    run_query(sql % tuple(data))
```

PHP:
```
function save() {
    $this->destroy();
    $sql = "INSERT INTO invitadoevento
            (evento, invitado)";

    $data = array("");
    $tmpvar = 0;

    foreach($this->invitados as $invitado) {
        $sql .= ($tmpvar > 0) ? ", " : " VALUES ";
        $sql .= "(?, ?)";
        $data[0].= "ii";
        $data[] = "{$this->evento->evento_id}";
        $data[] = "{$invitado->invitado_id}";
        $tmpvar++;
    }

    DBObject::ejecutar($sql, $data);
}
```

El método destroy()

El método destroy() se encargará de destruir de forma masiva, todas las relaciones existentes con el compuesto:

Python:
```
def destroy():
    sql = """DELETE FROM invitadoevento
             WHERE evento = %i" % \
                 self.evento.evento_id
    run_query(sql)
```

PHP:
```
function destroy() {
    $sql = "DELETE FROM invitadoevento
            WHERE evento = ?";
    $data = array("i",
                  "{$this->evento->evento_id}");
    DBObject::ejecutar($sql, $data);
}
```

El método get()

El método get() del conector lógico, realizará la consulta de forma similar a un método get auxiliar. Sin embargo, su comportamiento, será distinto. Con los resultados obtenidos de la consulta, deberá crear los compositores -al igual que el método get() de un objeto relacional simple- de forma iterativa y agregarlos al compuesto:

Python:
```
def get_relacion(self):
    sql = """SELECT invitado
             FROM invitadoevento
             WHERE evento = %i""" % \
                 self.evento.evento_id

    results = run_query(sql)

    for field in results:
        invitado = Invitado()
        invitado.invitado_id = field[0]
        invitado.get()
        self.evento.add_invitado(invitado)
```

PHP:

```
function get() {
    $sql = "SELECT invitado
            FROM invitadoevento
            WHERE evento = ?";
    $data = array("i",
                  "{$this->evento->evento_id}");
    $fields = array("invitado" => "");
    $results = DBObject::ejecutar($sql, $data,
                                  $fields);
    foreach($results as $field) {
        $invitado = new Invitado();
        $invitado->invitado_id = $field['invitado'];
        $invitado->get();
        $this->evento->add_invitado($invitado);
    }
}
```

Sobre Eugenia Bahit

Arquitecta de Software especializada en tecnologías GLAMP (GNU/Linux, Apache, MySQL, Python y PHP). **Agile Coach** certificada por la **Universidad Tecnológica Nacional de Buenos Aires**. Fundadora de **Hackers & Developers Magazine** (www.hdmagazine.org).

Miembro de la **Free Software Foundation** (www.fsf.org) e integrante del equipo de **Debian Hackers** (www.debianhackers.net).

Sitio Web: www.eugeniabahit.com

